12-1-2012

Hadoop-cc (collaborative caching) in real time HDFS

Meenakshi Shrivastava

Hadoop-CC (Collaborative Caching) in Real Time HDFS
Thesis

by

Meenakshi Shrivastava

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Computer Science

Supervised by

Dr. Hans-Peter Bischof

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

December 2012

The thesis “Hadoop-CC(Collaborative Caching) in Real Time HDFS” by Meenakshi Shrivastava has been examined and approved by the following Examination Committee:

Dr. Hans-Peter Bischof
Professor
Thesis Committee Chair

Dr. Minseok Kwon
Associate Professor

Dr. James Heliotis
Professor

Dedication

This Thesis work is dedicated to my parents and sister

Acknowledgments

First, I would like to express my gratitude to my advisor Professor Hans-Peter Bischof for his constant guidance, support, encouragement, valuable inputs and suggestions on this amazing journey of Thesis. His guidance throughout my thesis work, helped me in exploring new ideas, learning new things and writing this Thesis. I would like to thank Dr. Minesok Kwon for being part of the Thesis committee as Reader and for his valuable feedback and suggestions on my Thesis report. I would also like to thank Dr. James Heliotis for his interest in my Thesis work and being part of the Thesis committee as an Observer.

Abstract

Data is being generated at an enormous rate, due to online activities and use of resources related to computing. To access and handle such enormous amount of data spread, distributed systems is an efficient mechanism. One such widely used distributed filesystem is Hadoop distributed filesystem (HDFS). HDFS follows a cluster approach in order to store huge amounts of data, it is scalable and works on low commodity. It uses MapReduce framework to perform analysis and carry computations parallely on these large data sets. Hadoop follows the master/slave architecture decoupling system metadata and application data where metadata is stored on dedicated server NameNode and application data on DataNodes.

In this thesis work, study was performed on Hadoop Architecture, behaviour of filesystem and MapReduce in detail and concluded that processing of MapReduce is slow which was further confirmed by initial analysis and experiments performed on default Hadoop configuration. It is known that accessing data from cache is much faster as compared to disk access. Collaborative caching is one such mechanism in which the cache distributed over the clients or dedicated servers or storage devices form a single cache to serve the requests. This mechanism helps in improving the performance, reducing access latency and increasing the throughput. This coupled with prefetching enhances the performance.

In order to enhance and improve the performance of MapReduce, the thesis proposes solution of new design of HDFS by introducing caching references, collaborative caching along with prefetching coupled with Modified-ARC cache replacement. Each of the DataNodes would have a dedicated Cache Manager to maintain information about its local cache, remote caches and follow cache replacement algorithm. Initial analysis led to conclusion that caching references too help in improving performance. Modified-ARC helps in organizing the cache in a different way as recent, frequent and history of evicted items which is a better cache replacement policy and improves the execution time and performance of MapReduce.The evaluation of the results were done by comparing the results obtained with that of default configuration in psuedo-distributed and fully distributed mode.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem statement

Technological advancement has led to introduction of lots of computing related resources and with advent use of these applications online has led to the production of enormous amount of data. A challenge involves faster retrieval of these resources along with high performance. An effective mechanism to achieve this goal is the use of distributed systems. Distributed systems are fault tolerant, scalable and should provide high availability which are achieved through clustered approach of these systems.

Hadoop is one such storage system following clustered approach that stores large data sets, is highly fault tolerant and has large throughput. It runs on commodity hardware and it's key feature being separation of metadata from actual data. Hadoop's filesystem architecture and data computational paradigm has been inspired by Google File System and Google's MapReduce[3]. MapReduce paradigm[20] helps us perform analysis, transformations and computations on these massive amounts of data. Over the years, Hadoop has gained importance because of its scalability, reliablity, high throughput, analysis and large computations on these massive amounts of data. It is being used by all the leading industries like the Amazon, Google, Facebook, Yahoo. At Yahoo, there is a of span 25,000 servers, and stores 25 petabytes of application data, with the largest cluster being 3500 servers[11]. In a paper presented at Sigmod [2], describes how Facebook is using Hadoop in real time, with only few modifications made to it, it provides high throughput and low

latency. In 2010, it reported around 21 PB of data storage, in 2011 30 PB and in 2012 the data storage has crossed over 100PB[3].

Although Hadoop is widely used and popular today with its MapReduce paradigm being used in analysis and computation of large amount of data, but processing of overall execution of MapReduce task is slow. According to recent summit by Cloudera in June 2012, *Optimizing MapReduce Job Performance* [4], clearly shows that there are certain optimizations necessary in order to improve the processing of MapReduce task. Also in the paper [5], *Optimizing Hadoop for the cluster* by Christer, mentions that the default configuration is slow and optimizations are needed. In a paper [6], *The Performance of MapReduce: An In-depth Study* clearly mentions that MapReduce is slow, the processing execution time can be improved by adding more nodes to the cluster, but that is not a cost effective solution.My initial experiments and analysis on pseudo-distributed configuration and cluster configuration led to this conclusion that overall time taken to execute a MapReduce task is large. One of the key reasons for this is streaming of required data for executing the task is from the disk and due to more Rack-local tasks.

1.2 Collaborative Caching and Prefetching

It is known fact since years that accessing data from memory or cache is faster as compared to disk I/O's. For these data-intensive computations, focus has been more on processing of data rather than speeding up the process. Hence disk access has not been taken into consideration. According to [7], the initial phase which is map phase in MapReduce involves reading raw data from the disk and this task is I/O intensive. Caching data in such a scenario will help improve the execution time of the task hence improving the performance.

A cache hit would lead to faster processing of the request, but a cache miss would lead to disk access. Instead of disk I/O if we retrieve the required data from the neighbouring caches or remote memory, this mechanism is called *collaborative caching*. Caches of all

the participating machines taken together form a single cache or *global cache* and requests are satisfied from the remote caches instead of local disk access. It helps to improve the overall performance of the distributed systems and overcome the latency problem caused by the slower disk I/O [8]. Also as discussed in [8], collaborative caching helps in further enhancing the performance of the system.

Prefetching is another aspect and paradigm to overall performance improvement. There are many algorithms which define on what data to prefetch and how to prefetch. In prefetching, we already have the required data in cache beforehand and hence saves the time of retrieving from the disk.Hence collaborative caching along with prefetching helps in reducing the access latency and improving the overall system performance.

Another aspect leading to improvement is good caching policy ant it's combination with good replacement algorithm. LRU is not the optimal replacement algorithm[9]. Other hybrid schemes are now being used like FIFO-LRU, 2Q, MQ, ARC etc. Good replacement algorithm helps us ensure that we do not loose valuable blocks thereby improving the efficiency of the system.

1.3 My Work

As we have known and as discussed above that collaborative caching along with prefetching would definitely reduce access latencey, improve and enhance the system performance and that coupled with a good replacement algorithm would increase the efficiency. With these kind of high performance systems where achieving high performance is of essence, this scheme will help in lowering the execution times. In my thesis work, I have proposed collaborative caching scheme along with prefetching on data servers that is DataNodes from where actual raw data is read by the MapReduce task. It would have Modified-ARC as the replacement algorithm which is better replacement policy as compared to LRU. In case of maintaining the information regarding remote caches, minimal information of just

references would be cached which will help us determine which neighbouring node has data cached.

Modified-ARC defines better way of organizing cache by tracking the recent, frequent and evicted items. Each of the DataNode would have their dedicated Cache Manager who will manage the local caches, remote caches and cache replacement policy. For MapReduce tasks, there are many cases when the task to be carried out is not data-local, can be rack local or altogether in a different rack. In such cases to execute the task, data is streamed from another DataNode where the data resides which leads to disk I/O and n/w I/O. In such cases, reduced access latency can be achieved if data is being streamed from another cache. My initial experiments and analysis led to conclusion that caching the file references that make up block up also improves the performance. Upon request when system needs to lookup for these files, there is a disk I/O along with increase in lookup time because of the large data stored. If these file references are cached then lookup time reduces leading to overall improvement in the time.

1.4 Organization of Thesis

The organization of Thesis is as follows:
Chapter 2 explores the hadoop architecture which details about the components and their functioning, interaction among them and the different protocols it uses to interact. Chapter 3. goes in depth to explain the MapReduce job, its execution, working example of MapReduce job and explanation of the output of a typical MapReduce job. Chapter 4 explains the related work. Chapter 5 explains in detail the proposed design of Hadoop-CC (Collaborative Caching) and what factors make the thesis hypothesis hold true. Chapter 6 focuses on the implementation, explaining interaction of components of the proposed design, data structures used and how does it improve the overall system. Chapter 7 compares my approach versus the approach mentioned in the related work. Chapter 8 details about the analysis and conclusions made as part of studying and performing experiments on default

configuration of Hadoop distributed filesystem. Chapter 9, explains about how the evaluation was done of the new proposed architecture, assumptions, metrics, graphs obtained as a result of experiments carried out. Chapter 10, concludes the thesis. Bibliography.Chapter 11, explains the code and Chapter 12 refers to installation of Hadoop and commonly used commands.

Chapter 2

Hadoop Architecture

2.1 Architectural Components

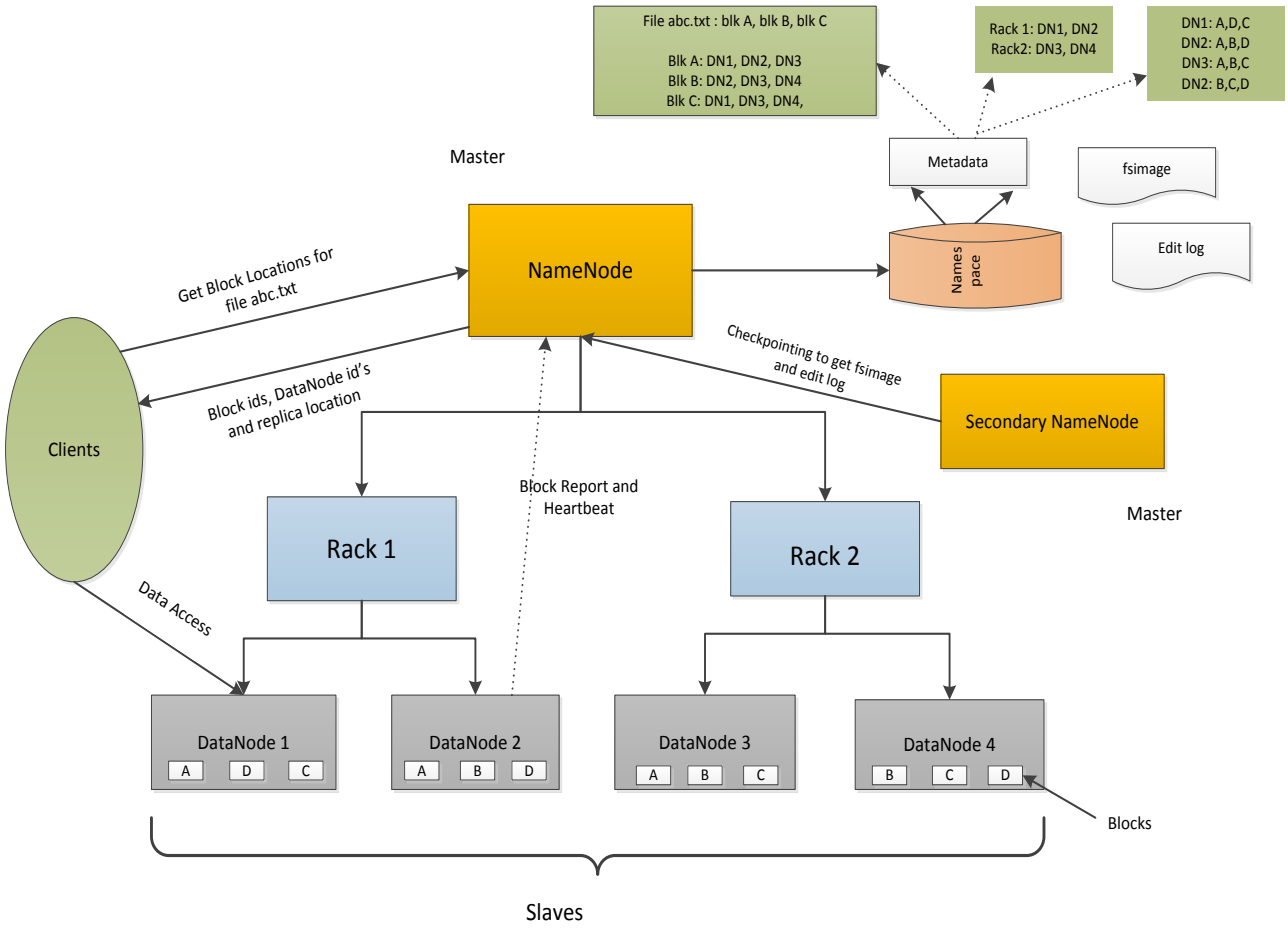

Figure 2.1: Hadoop Architecture, based on [11], [19]

Hadoop's MapReduce paradigm has been chosen as a candidate for implementation

and testing of new collaborative caching design. Since it is being widely used in industry including leading organizations such as Amazon, Google, Facebook and serves as one of the biggest cluster based distributed systems. It allows performing parallel computing and analysis on large amounts of data. Hence improving the efficiency and overall good performance of the system is a key factor. Installation of Hadoop is easier across any operating system like Windows, Linux, as its framework written in Java.

Figure 2.1 shows architecture of Hadoop Distributed Filesytem. It contains following components NameNode, DataNodes placed on racks, DFSClients and Secondary NameNode. It has a single master and many slaves architecture. The diagram also illustrates about the checkpoint process by secondary NameNode, DataNodes sending block reports and heartbeats, the example of metadata and mappings that NameNode holds in namespace. Each of the components are explained in detail below:

2.1.1 NameNode and Secondary NameNode

NameNode Namenode is known as the master or the central server of the Hadoop Distributed File System. NameNode maintains the namespace and metadata for the whole cluster by keeping the namespace in memory. It's other responsibilities include serving requests of DFSClients, monitoring health of the whole system and instructing DataNodes. An upto date block report from DataNode helps NameNode in building the filesystem and block datanode mapping[12]. It maintains two very important mappings, **mapping of filename to different blocks that makes the file and mapping of block to its location on different DataNodes**. Also it maintains the first mapping information on disk with the help of two files fsimage file and edit log. As seen in figure 2.1, it can be observed that File abc.txt is made up of blocks blk A,blk B and blk C. and mapping of which DataNodes the blocks reside on. Also in the figure it can be seen that from the block report received from DataNodes, NameNode knows about all the blocks that reside on a particular DataNode. If NameNode does not receive heartbeat from a particular DataNode then it assumes that the

DataNode is down and all the blocks on it are made unavailable. NameNode depending on the block report that was received earlier knows which blocks are unavailable and then it tries to replicate those blocks on other DataNodes. It also takes care of balancing the load and maintaining the replication factor for the blocks. While replicating, care is taken that rack awareness policy[1] is followed that two copies of block in the same rack and the third replica on other rack. Whenever namenode is formatted, it receives a new namespace ID and consequently all the DataNodes are given the same namespace ID.

Secondary NameNode Secondary NameNode connects to NameNode periodically and creates checkpoints for the metadata kept in memory by NameNode. It creates checkpoints by combining the information from edit log and fsimage from NameNode. These files are first created whenever NameNode is formatted. Secondary NameNode requests NameNode to create new edit log file where new operations can be recorded and preserve the old edits file untill new checkpoint file is available. Secondary NameNode applies all the operations listed in edits file to create a new fsimage file which is the checkpoint file. It then copies this checkpoint file back to NameNode while preserving a copy. Old edits file is replaced by new one and checkpoint time is updated. All these operations occur through HTTP requests and response. Checkpoints occur every hour by default but it is configurable parameter ”fs.checkpoint.period” or checkpoints take place when the edit log reaches size limit of 64 MB[10]. In event of NameNode failure these files from Secondary NameNode can used to get NameNode back from where it stopped functioning.

2.1.2 DataNode

DataNodes are data storage nodes storing massive amounts of data in the form of blocks. These nodes are spread across may racks and the racks are connected to each other through switches. The DataNode maintains an open socket server for clients to read and write data efficiently [12]. DataNode may also copy blocks, delete blocks and replicate blocks upon instruction from NameNode and can also contact other DataNode during task execution

and for replication. During startup DataNode performs a handshake to verify the namespace ID's consistency, an inconsistent namespace ID causes the DataNode to shut down. A DataNode interacts with NameNode in the form of heartbeats using TCP handshake to let NameNode know that it is alive whose heartbeat interval is 3 seconds[10]. Every hour it provides block report providing upto date information about the blocks stored which helps in building the namespace and recording any blocks which are under replicated. The blocks are replicated and stored on different DataNodes following the rack awareness policy. NameNode cannot contact DataNode directly but can only respond to heartbeat with commands. DataNode just maintains one mapping of **block to no of bytes**. The block report interval is scheduled for 1 hour which is configurable. At startup, DataNode sends a block report so that NameNode gathers data and which can help in building up the namesystem.

2.1.3 DFSClients

DFSClients connects to Hadoop FileSystem to perform basic tasks such as read, write, create on filesystem, appending data. Since Hadoop follows WORM architecture[11], it allows only appending to existing files. Clients also perform the function of submitting the MapReduce job, defining how they want the processing of the data to be done and review the results once it is done. They read the data in a sequential fashion from the DataNodes. For all the I/O operations, it uses a buffer size of 4 KB.

2.1.4 Namespace, FSImage and Edit log

Namespace is maintained by NameNode and it maintains an image of whole namespace which also includes blocks mapping in memory. Any changes made to this namespce is recorded. Namespace can be imagined as hierarchy of directory structure and files[10]. Operations such as read, write, creating new directories, files, renaming, open, close are allowed. NameNode maintains a log called Edit log, where any changes made to the metadata are recorded. For example creating a new directory, changing the name of the file[10].

The properties of filesystem, block mappings, filesystem namespace are all stored in a image file called FSImage. This file is also stored on the local filesystem. These files are read during NameNode startup and during checkpoint.

2.2 Different Operation Modes, Configuration Modes of Hadoop and Access

2.2.1 Operation Modes

NameNode operates in three modes:

Safe Mode When NameNode starts, it enters into Safe Mode where fsimage file and edit log files are read, edit log operations are applied to fsimage kept in memory[10]. This new fsimage file is now kept in memory along with new edit log file simultaneously flushing the fsimage file to the disk and disregarding the old edit file as operations have been applied. While in Safe Mode, read only view of the file system is provided to the clients[10]. During this mode it also receives the block reports from DataNodes to help build the namespace. It does not carry out any operations during this mode.

Normal Mode Normal operation mode of NameNode.

Backup Mode When NameNode operates in backup mode, Secondary NameNode is creating checkpoints, so that if failure occurs, recovery of NameNode can be made without any loss of data.

2.2.2 Configuration Modes

Hadoop can be configured to run in three modes:

Standalone A non-distributed mode and all the processes run in a single JVM [21].

Pseudo Distributed Here all the hadoop daemon run as a separate Java process but on a single local machine.

Fully Distributed The hadoop daemons run on cluster of machines.[10]

2.2.3 Accessing Hadoop Files

Psuedo-Distributed Mode To access any file on HDFS when it is configured to run on Pseudo-distributed mode is as follows: hdfs://localhost:54310/user/meenakshi/directory-name/abc.txt

Fully-Distributed Mode To access any file on HDFS when it is configured to run in Fully-Distributed mode is as follows: hdfs://namenode:54310/user/meenakshi/directory-name/abc.txt

2.3 HDFS Storage

In Hadoop Distributed Filesystem, files are stored in the form of smallest unit blocks. When an input file is copied from local filesystem to HDFS filesystem, the file is broken down into chunks. These chunks are known as blocks. For HDFS, this block size is quite large, where the default being 64 MB. This is configurable and can be changed by configuring the parameter dfs.block.size in core-site.xml configuration file. In order to ensure HDFS being fault tolerant, blocks are replicated across all the DataNode servers, and the default replication number is 3. The blocks in HDFS preserves the rack-awareness policy such that two copies of a block are placed in the same rack while other copy is place in a different rack[1].

2.4 Interaction among Components

Whenever DFSClient wants to read a file, it calls an open() on Filesystem object along with the index that is the offset from where it wants to read[10]. This filesystem object for

HDFS is distributed filesystem (DFS) object which contacts NameNode to get the blockid, DataNodes and its replica's address. The order of these DataNodes are sorted in the order, considering the DataNode closest to the DFSClient. The DFS object returns an inputstream of FSDataInputStream which in turn is wrapped in a object of DFSInputStream. The DF-

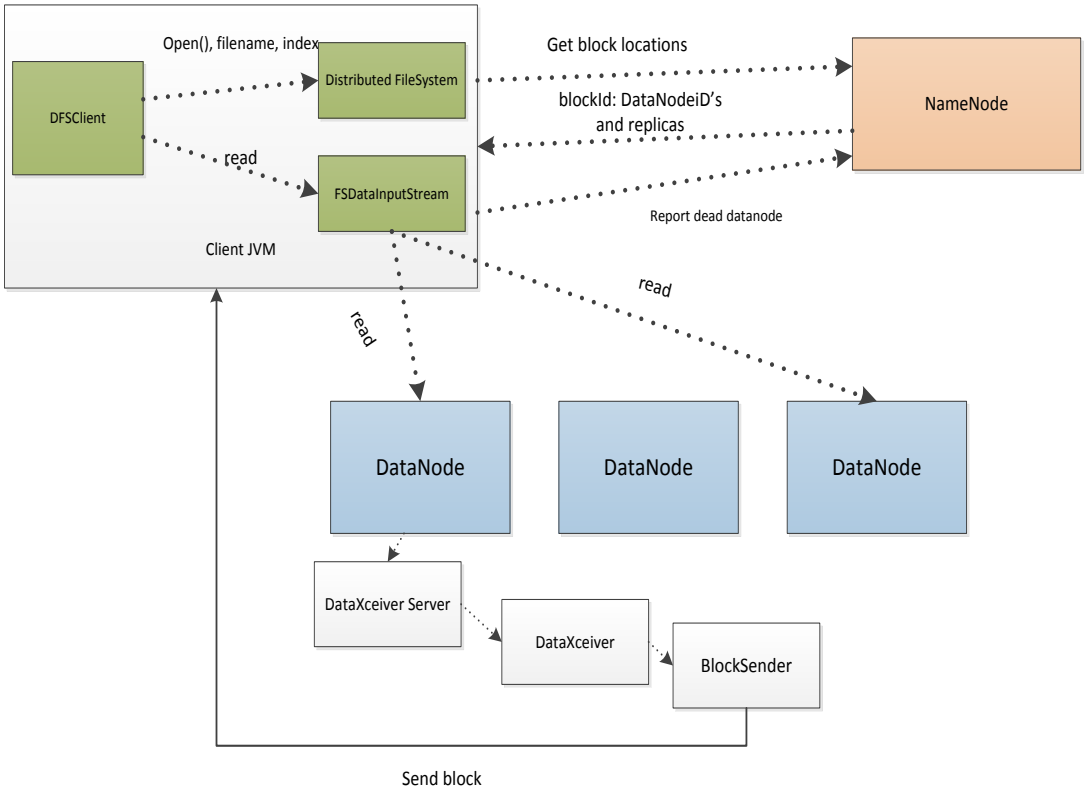

Figure 2.2: Interaction among components in Hadoop Architecture, based on [10]

SClient calls read on this stream for the first DataNode listed for the first block of the file. The DataNode has a DataXeciver server which is a daemon thread who continously listens to requests from either clients or other DataNodes and accordingly helps DataNode in sending or receiving block of data. The DataXceiverServer has DataXceiver thread running which helps in processing incoming and outgoing datastream and determines what kind of operation client wants to perform like read, write, copy. The DataXceiver in turn calls

BlockSender who sends data to the recipient in the form of chunks.The read operation is repeatedly called untill all the data for that block is read and thereafter the connection is closed. The DFSclient then moves to next DataNode for the next block. In case, DFSClient is not able to connect to the first DataNode listed for the block after two attempts, it adds the DataNode to deadlist. After all the blocks are read DFSClient forwards the deadlist to the NameNode, so that valid action is taken by NameNode.

2.5 Communication Protocols

Different components connect to each other using different protocols. Client and NameNode interact using ClientProtocol where client tries to establish an RPC connection with NameNode. The functions for ClientProtocol are create file, append to file, add a block, close, read, error, lease management. NameNode and DataNode interact using DataNodeProtocol.Since NameNode never directly contacts DataNode, the initiation is done by DataNode. It informs NameNode about its current state through DataNode registration, heartbeat, block reports. Client and DataNode interact using DataTransferProtocol for reading or writing purposes. This is not a RPC connection, but a I/O streaming using socket called the streaming protocol[12]. It's various functions include read block, write block and copy block. SecondaryNamenode and NameNode interact using NameNodeProtocol. DataNodes contact among themselves using InterDatanode protocol.

2.6 Drawbacks

2.6.1 Performance and Availability

NameNode is a dedicated server that maintains the namespace and carries out all the operations such as mapping of blocks, block operations, maintaining namespace, serving client requests, instructing DataNodes. With addition of more DataNodes, a threshold would arise beyond which the performance of NameNode starts degrading. Since NameNode is a single point of failure, if NameNode is down, the whole filesystem has to be taken offline.

To restore NameNode, Secondary NameNode's fsimage and edit log are used which are checkpoint files. This restoration may take time, if the namespace is large.

2.6.2 Scalability

HDFS keeps all the namespace and block locations in memory but the heap size of NameNode is limited. Also the entire cluster depends on the dedicated server NameNode for all its operations and namespace. Shvaock [11] gives an example of a large cluster with 3500 nodes which has 63 million blocks and since each block is replicated 3 times each DataNode will host 54000 block replicas. According to [11] Facebook and yahoo have 20PB of data. For a cluster size of 20PB, 30K clients requesting service from a single NameNode increases the overall workload on NameNode. Increasing DataNodes is directly proportional to the increase of workload. To add to it, overall memory size of NameNode is limited. Taking into consideration all these factors scalability of NameNode is limited.

2.6.3 Caching

Caching helps in improving the overall performance of the system especially when large number of files are to processed. Hadoop Distributed System does not support caching of blocks at the DataNode level, instead streaming of the blocks is done from the disk which increase the overall access latency and makes the system slower.

Chapter 3

MapReduce in Hadoop

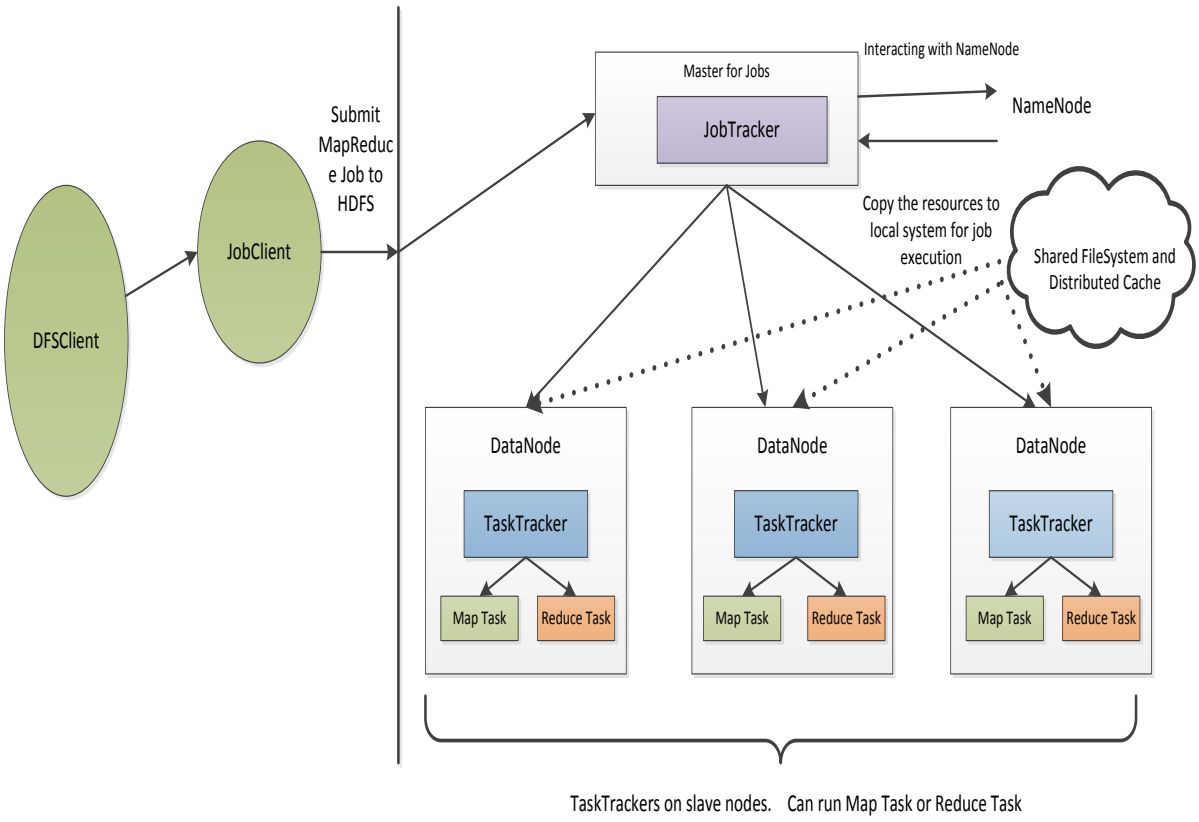

Figure 3.1: MapReduce Architecture, based on [10],[19]

To perform analysis and processing of stored data in HDFS, MapReduce paradigm is

used. MapReduce is parallel processing framework which divides the input into smaller inputs and carry tasks on it simultaneously hence achieving higher performance[10].

3.1 Architecture

The above diagram depicts the different components in MapReduce framework that participate in order to execute a task. The four main entities of MapReduce job are JobTracker, TaskTracker, JobClient, Distributed FileSystem/Distributed Cache. There is one master which is JobTracker and many slaves for carrying out the tasks which are TaskTrackers, one on each slave node (DataNode). Figure 3.1 depicts the placement of JobTracker and TaskTrackers, JobTracker's interaction with NameNode, JobClient submitting the job to HDFS, shared resources in common area of distributed cache, TaskTrackers running on DataNode.

JobTracker JobClient submits the job to JobTracker which serves as an interface for running MapReduce tasks. It assigns tasks to TaskTracker whenever there are empty slots as reported by TaskTracker and monitors TaskTrackers. For carrying out any tasks, JobTracker tries to find a TaskTracker with free slot such as it is data local or on the same rack. Since TaskTracker interacts with JobTracker in the form of heartbeats, if TaskTracker does not send heartbeat within a specified interval then it is considered that TaskTracker is dead and the task has failed. It then tries to schedule the same task on a different TaskTracker. If a particular task could not be completed successfully then TaskTracker notifies JobTracker of failure and in such a case JobTracker may schedule the same task on a different node and TaskTracker is blacklisted[10]. While scheduling any task, JobTracker consults NameNode to ensure about the data locality of the task. The above scenario can be observed in Figure 3.1 where there is a single JobTracker and many TaskTrackers. JobTracker can run on same machine as NameNode or can run on a different machine as well.

TaskTracker It is a node which starts the MapReduce tasks, monitors it, contacts JobTracker for new tasks and notify results[10]. TaskTrackers send heartbeats messages to JobTracker to indicate that they are alive and and as a part of the message also indicate if it has free slots to carry out the tasks. If TaskTracker is ready to accept new tasks, then accordingly JobTracker assigns a task which is return value as part of heartbeat. Every TaskTracker has certain fixed number of slots which tells the maximum tasks it can run. When scheduling a task, JobTracker tries to find a empty slot on the same DataNode on which the input resides. If that fails then it tries scheduling a task on machine which resides on the same rack. To carry out any task, TaskTracker spawns new JVM everytime so that if the task fails, it does not bring down TaskTracker. These tasks are monitored by TaskTracker and notifies the JobTracker when the task completes. As seen figure 3.1, TaskTracker can carry out either of the of the two tasks which is Map Task and Reduce Task and there are TaskTrackers running, one on each slave node

JobClient JobClient provides an interface for the users to submit jobs and interact with JobTracker. It also tracks job's progress and notifies about it to the user by outputting on the console and when requested can provide the necessary information regarding cluster. Whenever a job is submitted, input directory and output directories are checked if they exist, input splits are calculated, distributed cache is set up for that job and all the necessary resources for carrying out the task are copied to local location where the particular task will be executed.

Distributed Cache It is used to distribute the necessary files to slave nodes before the start of execution of any scheduled task. The files can be of any form like the jars, archives. These are files specified by the application and are copied only once. Such files are unjared at the slave nodes.

Tasks There are main two types of tasks namely, Map tasks and Reduce tasks. Map tasks in turn is subdivided into two phases, map phase and sort phase. Reduce tasks are

subdivided into three phases which are copy, sort and reduce phases. Whenever an input is provided for the mapreduce task, the input is divided into chunks what is called the splits and mostly the size of it is that of the block size.

3.2 Job Execution

A MapReduce job execution goes through series of steps in order for the task to be assigned, start running and outputting its progress. JobTracker monitors all the tasks across the cluster, while TaskTracker actually carries out the tasks. Steps are explained as follows:

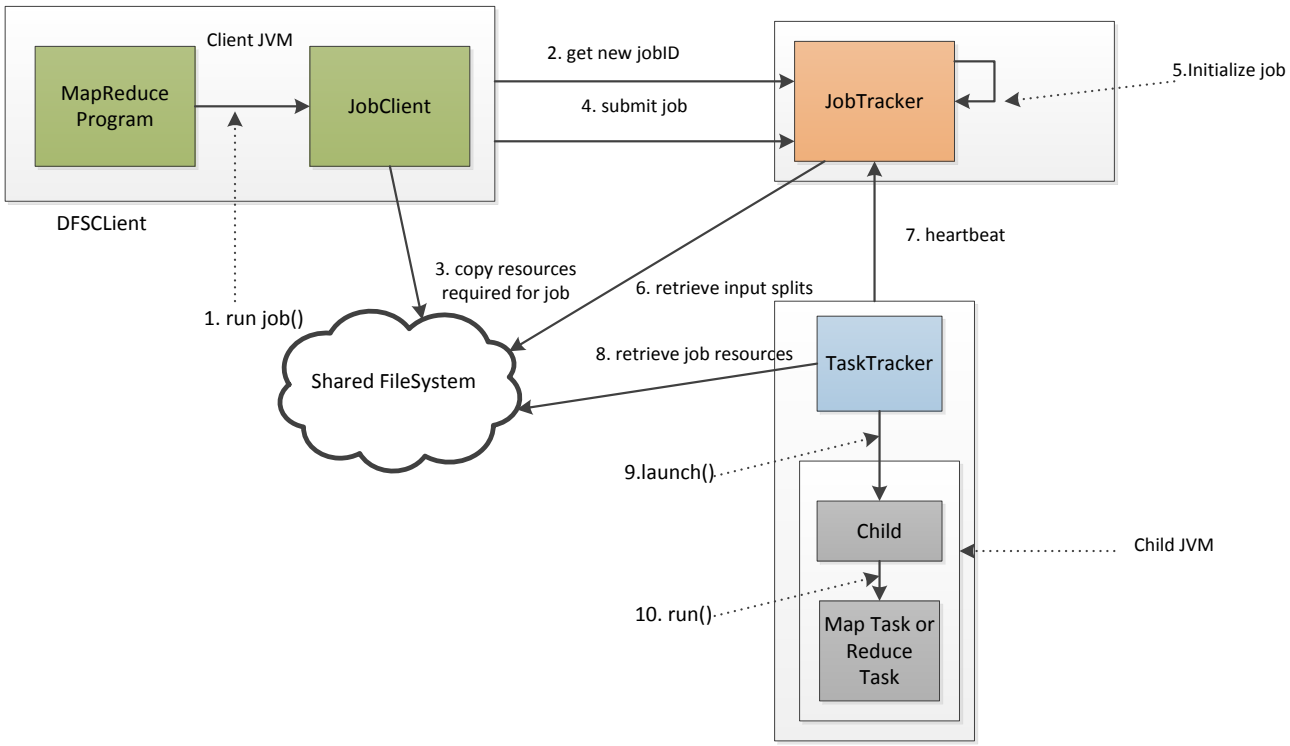

Figure 3.2: Job Execution Architecture [10]

- Initially runjob() is called on JobClient which creates a new instance of JobClient

which in turns submits the job. Once the job is submitted, its progress is monitored and reported to the console.

- Once the job is submitted, JobClient gets a new jobID from JobTracker, checks for the ouput directory and calculates the number of splits.
- JobClient copies the resources required for the execution of the task to the shared filesystem or the Distributed Cache.
- Now submitJob() is called on JobTracker which indicates that job is ready for execution and is added to the queue. JobScheduler will pick up the task and monitor the progress of the task.
- JobTracker initializes the job.
- To start the tasks, JobScheduler needs to know the input splits which are retrieved from shared filesystem, which JobClient created while submitting the job. The number of map tasks are calculated which equals the number of splits computed. No. of reducers is determined by the property mapred.reduce.tasks and all the scheduled tasks are given task id so as to monitor its progress.
- Whenever TaskTracker sends heartbeat messages, the message contains whether it has empty slot or no. If it has empty slot then JobTracker assigns tasks. By default it has two slots for map and two for reduce. For map task, it takes into consideration the locality that is it will assign a map task to the TaskTracker, considering the input split closest to TaskTracker. First preference is given to assigning tasks which are data local, if that is not possible, then assigning tasks such that it resides in the same rack and last data residing in the other rack.
- After the task has been assigned, it copies all the necessary files locally from distributed cache, unjars the archives to run the task and a new TaskRunner instance is created.

- To run this task, TaskRunner spawns a new JVM which is the child process to start running the task which may be Map or Reduce. A new JVM is launched, so that any errors in map function and reduce function does not affect the running of Task-Tracker. The JVM can be reused for other tasks. Umbilical Interface[10] is used by child process to indicate about the progress of the task.
- JobClient polls the JobTracker each second to get the status of the running job. When a job is complete and JobTracker receives a successful message, if JobClient polls, it learns about the completion of the task and displays the message to user console.

MapReduce Data Flow In the previous section we went through steps of how the task is assigned and when the task starts running. This section details of the different phases the task has to go through before we obtain the final output. The data processing on the input files are done with the help of two very important steps Map Task and the Reduce Task.

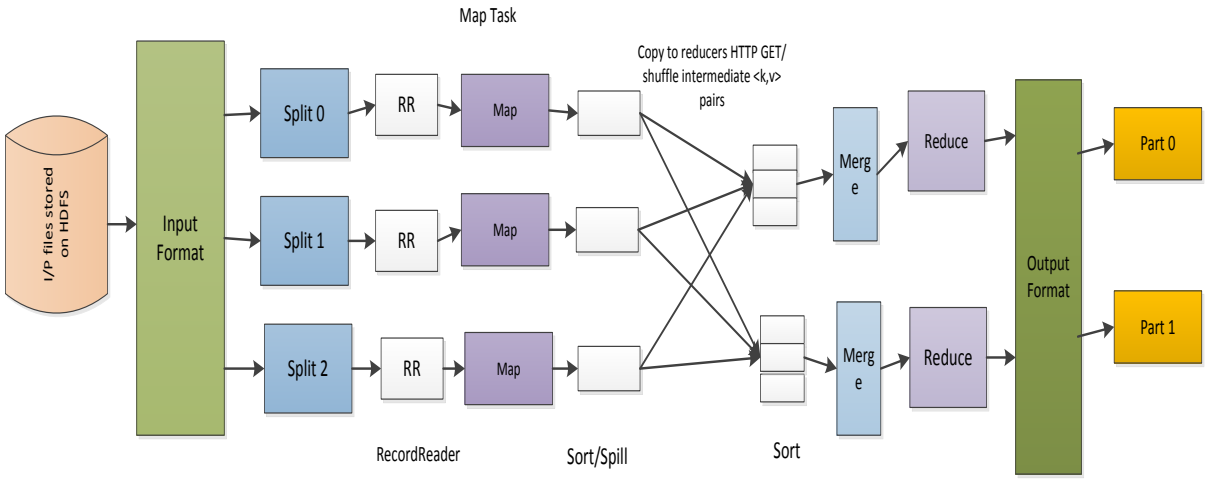

Figure 3.3: Data Flow of a MapReduce Task, based on [10], [13]

Input files are provided as an input to the MapReduce task and these files reside on

HDFS on DataNode servers. Initially we have to define the format of the input type which is defined under InputFormat. These files are broken down into smaller inputs which are called the input splits. The number of map tasks are equivalent to number of computed input splits that needs to be executed. The splits size is that of block size. These splits reside on different nodes and hence map tasks are executed on different nodes in parallel and accordingly process the input. The RecordReader takes in the input and converts it into key,value pair that can be read by Mapper, the first phase of MapReduce [13]. The RecordReader calls the Mapper. Mapper is where the user-defined map function defines how the processing is to be done for the input of key,value provided by the RecordReader. A seperate JVM process is launched for each Mapper Task. These key, value pairs will finally be forwarded to the reduce function. After the map tasks have completed processing they produce intermediate key, value pairs. Map phase consists of map phase and sort phase. As the map task progresses and produces the intermediate key, value pairs they are kept in memory buffer and and then sorted in the sort phase. Once the in-memory buffer fills up, it spills the output to the local disk and not to the HDFS filesytem[13]. The next phase is the Reduce Task.

The reduce task consists of three phases copy/shuffle, sort and reduce phase. The intermediate results produced by map phase are copied via HTTP GET to where the reducers are. This process is known as shuffling. These results copied on a node are again sorted on keys and similar keys are merged together before providing as an input to reduce phase. In Reduce phase, reduce function reduces the values associated with that particular key. The reduce function is also user-defined. It receives a key and a list of values associated with it. The resultant key, value pairs are written to local disk in output files. They are written in the format specified by the Output Format.

3.3 Example

This section explains in detail, how the map reduce works with the help of an example. The example explains the processing of a very well known ”WordCount” example as it

progresses through different phases. In a ”WordCount” problem, the aim is reading all the inputs and determining how many times a word occured in the input file.
First we need to have the input format . Let the InputFormat be TextInputFormat. Next we need the input splits. Let's consider for this example, we will create splits one per line. Suppose the sentences are

- ”I am a computer science student and I like programming”.
- ”I am a student at RIT”.

Since there are two splits, hence there will be two Map Tasks scheduled. The map function processes one line at a time and tokenizes into words, outputs a key,value pair like <word,1>. For this example we consider a single reducer.

Input splits are first processed using Map phase followed by Sort Phase.

3.3.1 Map Task

Map Phase Output for first Map <I,1>
<am,1>
<a,1>
<computer,1>
<science,1>
<student,1>
<and,1>
<I,1>
<like,1>
<programming,1>

Sort Phase for first Map Output from map phase enter sort phase, whose results obtained are as follows:

<a,1>

<am,1>

<and,1>

<computer,1>

<I,1>

<I,1>

<like,1>

<programming,1>

<science,1>

<student,1>

Map Phase Output for Second Map <I,1>

<am,1>

<a,1>

<student,1>

<at,1>

<RIT,1>

Sort Phase for second map Output from map phase enter sort phase, whose results obtained are as follows:

<a,1>

<am,1>

<at,1>

<I,1>

<RIT,1>

<student,1>

3.3.2 Reduce Phase

After the map phase is completed, it enters the reduce phase. A reduce phase has copy, sort/merge and then final reduce phase.

copy phase The map's intermediate key value pairs are copied to where the reducers are located. We are considering only a single reducer. So all the values from first as well as the second map go to a single reducer. After the intermediate values are copied, they enter the sort phase and merge phase.

Sort phase <a,1>
<a,1>
<am,1>
<am,1>
<and,1>
<at,1>
<computer,1>
<I,1>
<I,1>
<I,1>
<like,1>
<programming,1>
<RIT,1>
<science,1>
<student,1>
<student,1>

Merge Similar Keys are merged

<a,<1,1>>

<am,<1,1>>

<and,1>

<at,1>

<computer>

<I,<1,1,1>>

<like,1>

<programming,1>

<RIT,1>

<science,1>

<student,<1,1>>

Reduce phase In the reduce phase, values related to keys are added.

<a,2>

<am,2>

<and,1>

<at,1>

<computer>

<I,3>

<like,1>

<programming,1>

<RIT,1>

<science,1>

<student,2>

This is the final output.

3.4 Output Explanation

Following section explains about the output produced after the completion of mapreduce task execution. These parameters require scrutinizing in order to ensure that job run was successfull and we have received the expected output. This output is termed as counters and there are job counters, task counters, fileinput format counters, fileoutput format counters, filesystem counters [10].

3.4.1 Pseudo Distributed Mode

Below Figure 3.4 shows a screenshot of the output obtained after running MapReduce job in a pseudo-distributed environment.

Job Counters

- Launched reduce tasks : no of reduce tasks that are launched.
- SLOTS MILLIS MAPS : total execution time of all the maps taken together
- Launched map tasks : no of map tasks started for a given input
- Data-local map tasks : no. of map tasks launched on the same node as the input resides.
- SLOTS MILLIS REDUCES : total execution time of all the reduces taken together

Filesystem counters

- HDFS BYTES READ : bytes that were read from HDFS filesystem
- FILE BYTES READ : bytes that were read by local filesystem

FileInputFormat

- Bytes Read : the number of bytes read by all the map tasks using input format as fileinputformat.

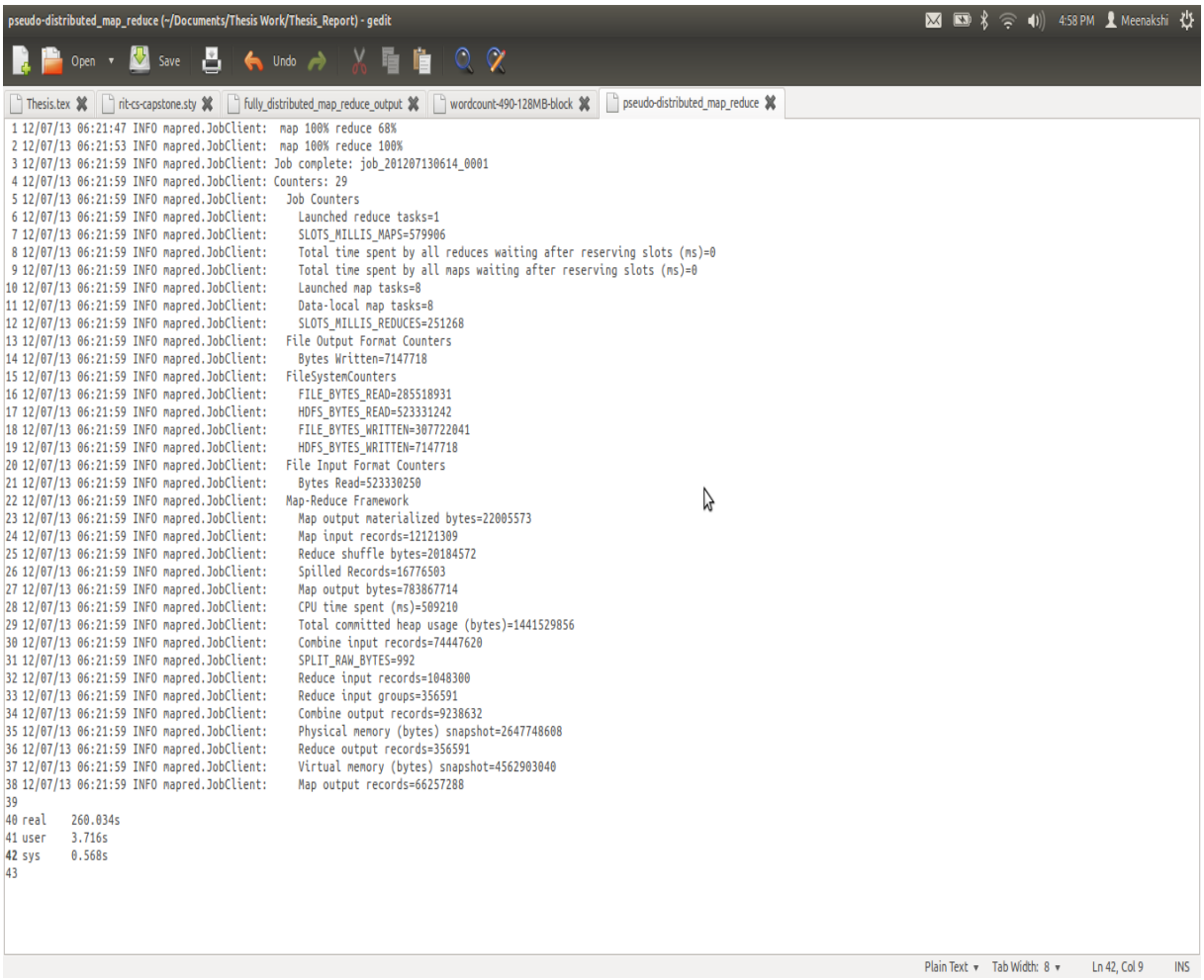

Figure 3.4: MapReduce output for Psuedo-Distributed Mode

Task Counters

- Map input records : records that were read by all the maps taken together, in an event whenever a record is read by RecordReader and passed to map
- Spill Records: total bytes that were spilled to the disk when map and reduce were executing
- Map input bytes: the number of uncompressed bytes that were read by all maps taken together
- SPLIT RAW BYTES: bytes that are read of split's metadata which is length, offset etc.

- COMBINE INPUT RECORDS: total input records that are read by all the combiners taken together.
- REDUCE INPUT RECORDS: total input records that are read by all the reducers taken together.
- CPU time spent: total time spent by CPU on a task
- Reduce shuffle bytes : no of bytes of map's output copied to the location where reducers are launched.

3.4.2 Fully Distributed Mode

Below Figure 3.5 shows a screenshot of the output obtained after running MapReduce job in a fully distributed environment. The only difference between the output of a pseudo distributed mode and fully distributed mode is in the following counters:

- Rack-local map tasks : no. of map tasks that were launched such that task was launched in the same rack as the input resided. In short, the input resides on a different node as the task was scheduled.
- Launched map tasks : total map tasks launched for a map phase
- Data-local map tasks : map tasks launched on a node such that input resides on the same node.

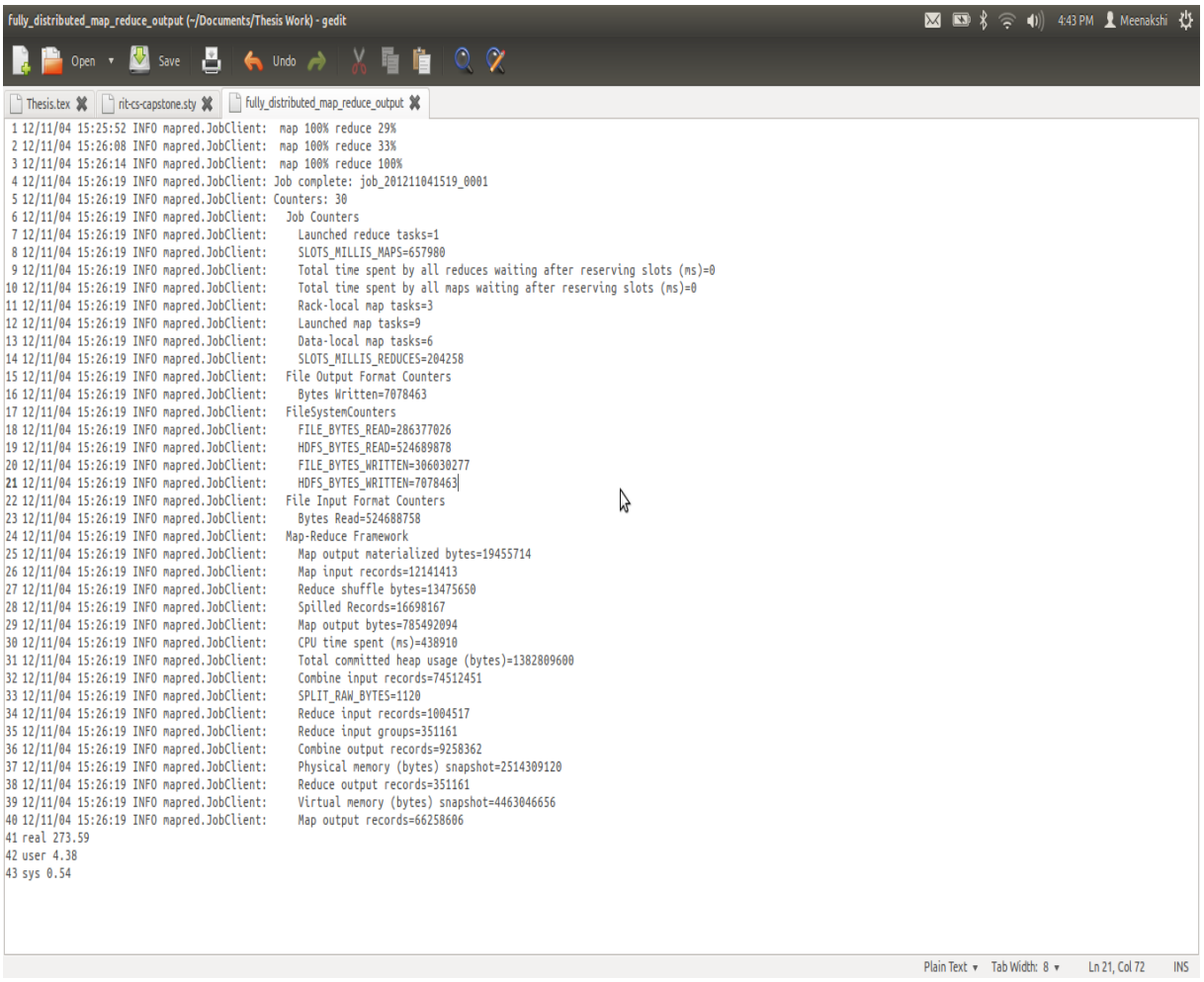

Figure 3.5: MapReduce Output for Fully-Distributed Mode

Chapter 4

Related Work

4.1 PACMan:Coordinated Memory Caching for Parallel Jobs

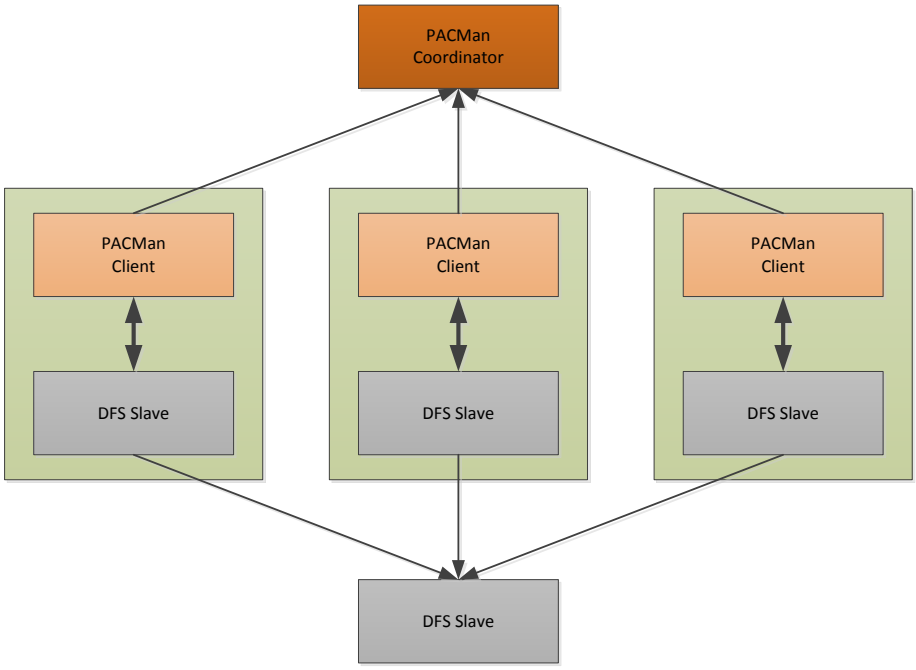

Figure 4.1: PACMan: Coordiante Memory Caching for Paralle Jobs, [7]

Figure 4.1 shows diagram of PACMan Coordinator integrated into slave machines.

According to PACMan, when multiple jobs are run in parallel, job's running time can be decreased only when all the inputs related to running a job are cached. So according to Dhruba[7] *et.al*, either cache all the inputs related to that particular job or do not cache the inputs at all. Caching only part of inputs will not help in improving the performance. These massive distributed clustered systems have large memories and job execution performance can be improved if these memories can be utilized to the fullest. The main aim of these distributed clustered systems are to break up a large job into many small jobs and an attempt is made to run these jobs in parallel. PACMan is a caching service that coordinates access to the distributed caches[7]. This service aims at minimizing total execution time of job by evicting those items whose inputs are not completely cached. For evicting the inputs which have been minimally used, LFU-F algorithm is used. Experiments were performed on Bing and Facebook and their results showed a reduce in the average completion time by 53% and 51% and improvement in the cluster efficiency by 47% and 54%[7].

According to them, localizing the job and caching all the inputs for a particular job will help in performance. They have used MapReduce and Dyrad as examples to illustrate their algorithm and hypothesis. According to them reading the raw input from filesystem is IO intensive and forms 79% of the phase[7]. They define a new parameter wave-width of the job which refers to total tasks which can be executed in parallel at a time. They conducted experiments to see how the memory-locality[7] tasks would perform and it was observed that job execution time improved. Since in Hadoop, the reduce task starts executing only after the map tasks are completed, they have conducted experiments on Hadoop and they observed that if the data is not available in cache then reduce tasks wait for a while before they can actually start executing. They emphasize on memory-locality tasks which is an important factor contributing to cluster efficiency. In short locality of jobs play a very important role.

Dhruba[7] *et.al* proposes an architecture called the PACMan which coordinates the caches globally and it takes care of two things which is support queries where block is cached and

coordinating the cache replacement[7]. Its architecture includes a coordinator service and PACManClients are located on nodes where data lies. Blocks are cached on these clients. Whenever there are any changes on these clients, they report to the coordinator. This coordinator holds mapping of blocks to location where the blocks are cached. It also includes the information regarding this block belonging to which file and wave-width of the file. This overall structure of the coordinator is used for scheduling tasks which are memory local, in implementing sticky policy[7] to check on the incomplete files. Also this information is used by LIFE(providing an improvement on the total completion of the job) and LFU-F[7]. PACMan clients on the nodes cache blocks, serve those blocks and evict those blocks when required. This caching mechanism is implemented on nodes where the data lies that is where the tasks are executed. Whenever any request comes in for any data, they first contact the PACManClient located on node to check if any data is cached. If the data is not cached then it gets the data from disk. It does not attempt to contact other clients to check if the data is available. Also it emphasizes to schedule a data local job. For replacement policies they go for global cache replacement policies which are LIFE and LFU-F[7]. The overall job execution time is reduced attempting to schedule jobs of smaller-wave widths.

4.2 Dyanmic Caching Mechanism for Hadoop

- The main aim defined in this paper is caching mechanism having modified replication policy, allowing concurrent access to the data and algorithms relating to locality of data which focuses on the decrease in the overall job completion time.

- Their caching mechanism is based on Memcached[16] which are a set servers storing the mapping of block-id to datanode-ids. These servers also serve the remote cache requests. The caching of blocks is carried out on DataNodes.

- The paper mentions about two different design architectures, the different ways to serve a request for receiving it. First architecture defines where DataNode is making

a request for block. At this point simultaneous request is sent to NameNode and Memcached servers. Reply is sent to the DataNode by NameNode as well as the Memcached servers, once both do lookups for the requested block. If DataNode receives a reply true from MemCache then block is accessed from cache, else block is accessed from the disk whose location is indicated by the NameNode. In the second design architecture, again DataNode request is sent to NameNode as well Memcached. But this time NameNode does not reply although it performs a lookup for the block. Memcached does a lookup too

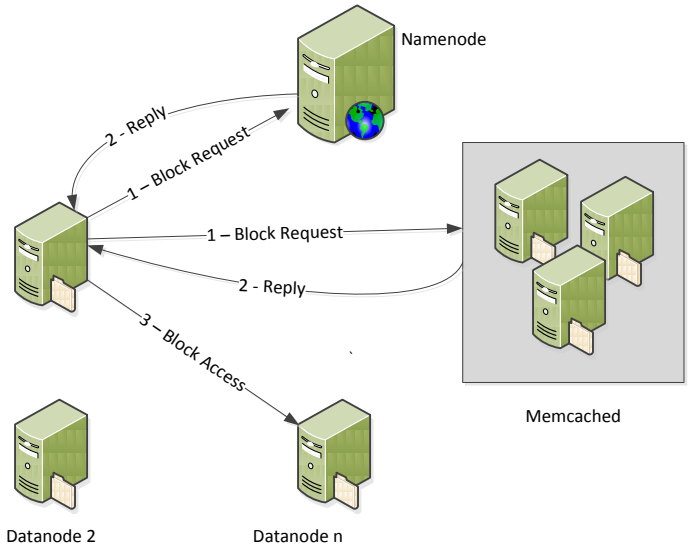

Figure 4.2: Architecture for Dynamic Caching Mechanism for Hadoop using Memcached servers,[16]

and replies back to DataNode. If Memcached does not find the block then it replies to both NameNode as well as DataNode. In such a case, NameNode sends the block locations to DataNode. The design also supports prefetching. Whenever a request for block is seen by Memcached, lookup is performed for neighbouring blocks as well. If neighbouring blocks are missing then Memcached requests NameNode to look for replicas and if available, corresponding DataNodes are requested to cache

those blocks and accordingly Memcached updates its locations.

4.3 Other Related Work

Lustre a high performance file-system which has features such as scalability and high availablity, required for a distributed file-system. It's architectural components are object storage targets, object storage servers, object disks and metadata servers. Lustre filesystem also seperates metadata from data. On request to modifications to a file, lock is preserved on the file. Lustre has COBD which is collaborative cache driver[15] used to reduce the overall read load and cache the data which is being accessed frequently. Client FS refers to client lesystem, OSC refers to Object Storage Client and WB refers to Write Back Cache. Collaborative cache activates multiple OSTs to cache frequently accessed data. Information about clients and cache server including the data cached by them is maintained by OST so that any read request by client can be served by OST redirecting the request to correct cache server[15]. On any request to serve data, lock is maintained. While the lock is being maintained, in the meantime OST looks up to see if the data is cached and accordingly the request is redirected. To make caching decision of which OST to select, decision is made by the referral module. If an OST having cached data is found, the OSC which is the client will be provided with the caching node UID. In a scenario where no caching data is found, still the request is forwarded to OST, a lookup is performed. No data found in cache causes OST to cache the data.

Chapter 5

Hadoop-CC (Collaborative Caching) - Proposed Design

5.1 Architecture

This chapter explains in detail the proposed architecture, algorithm and reason that led proposed architecture help in achieving the desired output of lowering the overall MapReduce job execution time. There has been change in the architecture as mentioned in my proposal and a new architecture is adopted. For sections below where changes have been made, a description is provided along with reason of this new architecture and modifications.

Fig 5.1 shows the diagram of overall proposed architecture for Hadoop-CC. To add the functionality of collaborative caching on DataNode a new process of Cache Manager has been introduced into DataNode. There are other additional functionalities on the components; DFSClient and the NameNode. Also new OP_Codes has been defined as the Data Protocol Transfer for errors and data transfer to the recipient during caching. Following explains Cache Manager functionality in detail, along with new functionalities added on DFSClient and NameNode.

5.1.1 Cache Manager

Each of the DataNode have their dedicated CacheManagers who have responsibilities of managing caches, lookup in local as well as global cache image upon request, replacement policy for cache, eviction policy for cache when cache is fully utilized. Each of the responsibilities are explained in detail as follows:

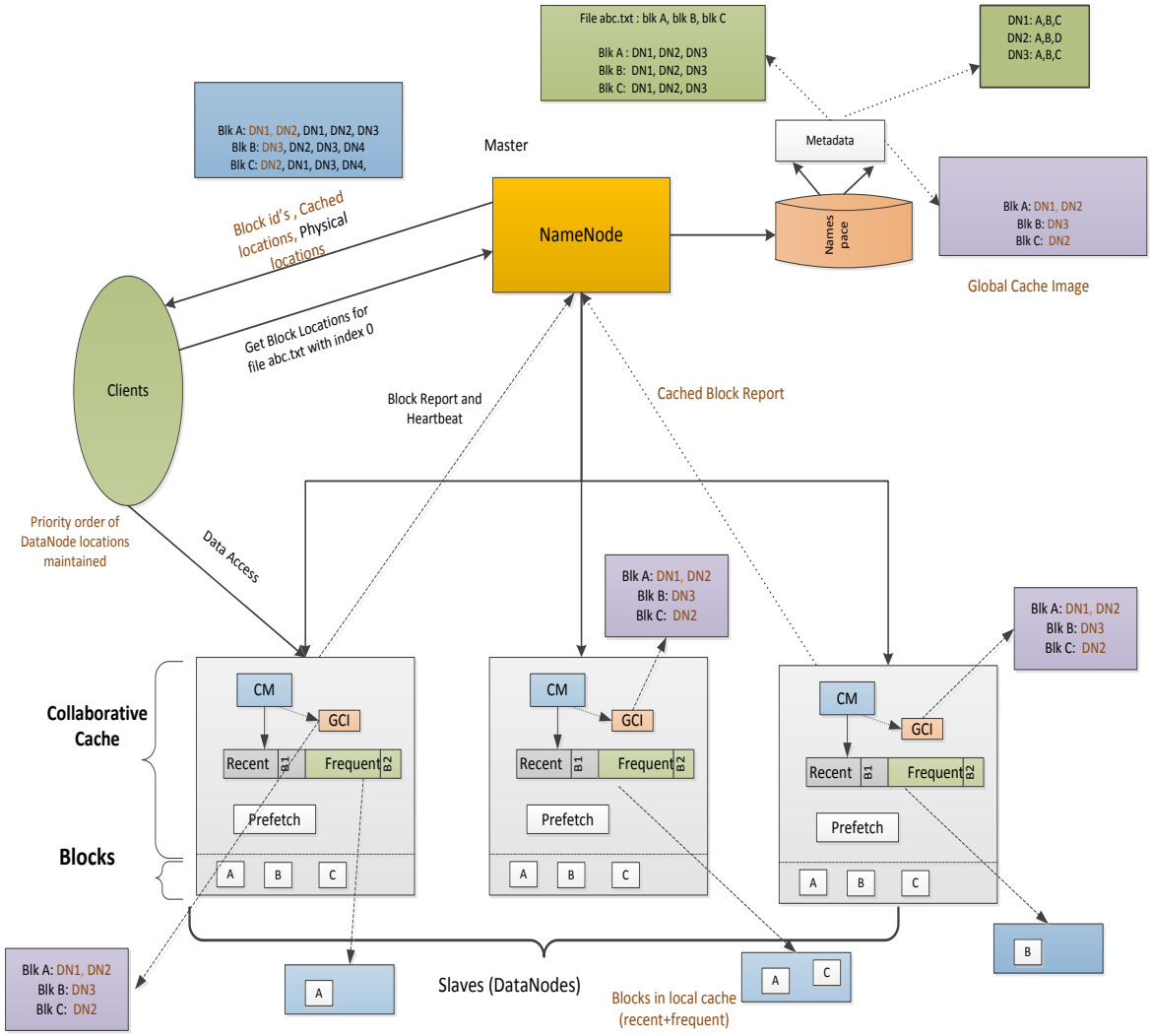

Figure 5.1: Proposed Architecture

5.1.2 Caching

The local cache is divided into four sections namely recent cache, frequent cache, recent history and frequent history. Logically we can say two cache sections where recent cache and recent history as one cache and frequent cache and frequent history as other. Total Local cache is comprised recent cache as well as frequent cache. As seen the diagram 5.1, each of the Cache Managers have their own local cache which is recent + frequent. DN1's local cache has just block A, where DN2's local cache has blocks A and block C.

Prefetching A prefetch buffer is maintained where the references of the blocks which are meta files and block files are cached. These references are cached when initially a request for the block comes in at the DataXceiver of DataNode.

Replacement A cached object will be replaced when the cache is fully utilized. There are three sets of replacement policies:

- Whenever a new block arrives to be cached in the recent cache, and the cache is fully utilized, in such case the LRU block is evicted from the cache, but its reference is placed in the recent history.
- A new block to be cached in the frequent cache and the frequent cache is fully utilized, in such a case the LRU block is evicted and its reference is placed in the Frequent history.
- A fully utilized Recent history or frequent history cache causes eviction of the references completely.

Eviction After series of experiments and considering performance, LRU cache eviction policy will be used for each of the individual caches which is recent, frequent, recent history, frequent history. This eviction policy will also be followed for the prefetch buffer.

As the blocks are evicted from caches, their references are placed in the history and as the history cache fills up eventually they are totally evicted from the cache.

5.1.3 Global Cache Image

Fig 5.2 shows the diagram of Global Cache Image. Global Cache Image lookup is done by Cache Manager upon local cache miss. RemoteCache list is also named as global cache image(GCI), as seen in the diagram comprises of mapping of block to DataNodes. This denotes that this block is cached on which DataNodes in the cluster. This mapping is maintained by NameNode and copy of it is sent to all the DataNodes as a response to cache block report. As shown in diagram 5.1, each of the DataNodes have GCI and it indicates that if we are at DN3 and we want to perform a look for block A, then it is cached at DN1 and DN2.

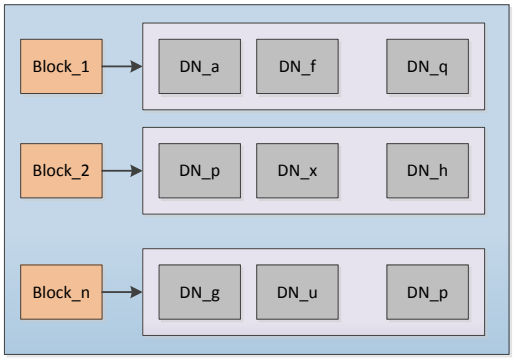

Figure 5.2: Global Cache Image

5.1.4 Difference between previous proposed Architecture

- **Block Forwarding**: CacheManager does have the support for Block Forwarding. The reason this was discarded because streaming data across the network takes time and with huge block size, it reduces the overall performance of the system.

- **Cache Manager Responsibilities**: It does not take care of singlets, since that would involve Block Forwarding and Block Forwarding is not considered due to performance reasons.

- **Replacement**: The entries are evicted as defined in proposal but the cache is not adaptable to increase the size of the individual caches. Since with the current simple design which involves taking the idea of ARC and adding my modifications of prefetch bufffer and caching references proved to be efficient and helped in better performance.

- **Eviction**: Each of the individual caches in the Modified-ARC was proposed to follow LRU-LFU-MFU which was to look for file references, replica references and block references. But the current implementation for each of the caches have LRU. The reason this has been chosen was, to keep the system more efficient and avoid lots of lookups.

- **Remote Cache List**: The reason previous remote cache list also known as Global Cache Image was updated, due to, two reasons. First reason, everything in the Hadoop system deals with block, hence using block as the key helps in better searching. Morever, total local cache is comprised of recent and frequent hence we need not differentiate it into recent and frequent. There is no block forwarding due to extra time incurred by forwarding the block over the network. This datastructure helps NameNode to better maintain Global Cache Image without extra cost.

5.1.5 NameNode Management

NameNode is the central co-ordinator for maintaining the Global Cache Image. It builds its global cache image when it obtains the cached block report from the DataNodes. As soon as it obtains its report, it updates the mapping of cachedblock to DataNode. Upon updation of the Global Cache Image, as response to cached block report it sends a copy of Global Cache Image to the DataNode via DNA_UPDATE_GCI command.

5.1.6 DFSClient

DFSClient has to contact NameNode for getting the block locations for a particular file. Hence to this request along with non cached block locations, cached block locations are also provided as a response. If it has received the cached block locations, then obtained list is sorted by the DFSClient so as to read from cached block.

5.1.7 DataNode

DataNode provides a cached block report of its local cache to NameNode after periodic interval. As a response to this report NameNode commands it to update global cache image on DataNode.

5.1.8 New OP_Codes

These OP_Codes have been introduced as part of the collaborative caching mechanism and as DataNode transfer protocol. These OP_Codes help in determining the next set of steps to taken by DFSClient or DataNode.

- **OP_READ_BLOCK_CACHED** : To indicate that DFSClient is attempting to read the data from cache.
- **OP_STATUS_BLOCK_CACHED_ELSEWHERE**: DataNode sends this to DFSClient to signify that this block is cached on a different node.
- **OP_STATUS_BLOCK_NOT_CACHED**: DataNode sends this to DFSClient to signify that this block is not cached at all.
- **DNA_UPDATE_GCI**: This is the instruction sent by NameNode to DataNode in response to the cached block report.

5.1.9 Terms

- RackLocal: The job is scheduled on a different node and input resides on a different node.
- DataLocal: The job is scheduled on the same machine as the input resides.

5.2 Proposed Algorithm

This section explains the overall collaborative caching algorithm implemented for Hadoop system along with Modified-ARC algorithm for caching policy and caching scheme.

5.2.1 Collaborative Caching and Prefetching on Hadoop

An approach proposed in order to improve the overall performance of execution of the MapReduce jobs. Collaborative Caching takes place on the DataNodes with the help of Cache Managers. This along with prefetching references and caching them helps in overall enhancement of the job execution timings. Here references refer to meta file and block file. A Modified-ARC algorithm is used as caching policy. Here simple collaborative scheme is followed where if there is a local cache miss, then DataNode notifies DFSClient of the next cached DataNode to connect if any, or to connect to a non caching DataNode.

5.2.2 Modified-ARC Algorithm

Fig 5.3 shows the diagram of a Modified-ARC. A variant of this algorithm is implemented. Basic idea is to divide the caches into two different sections namely cached objects and history objects. Cached section contains the actual data and History section contains the references. Hence the cached section is further divided into Recent Cache and its Recent History and Frequent Cache and its Frequent History.

- Recent Cache: A Recent cache is a cache where the block seen for the first time is placed.

- Frequent Cache: A second reference to the same block will cause the block to be placed in the frequent cache.

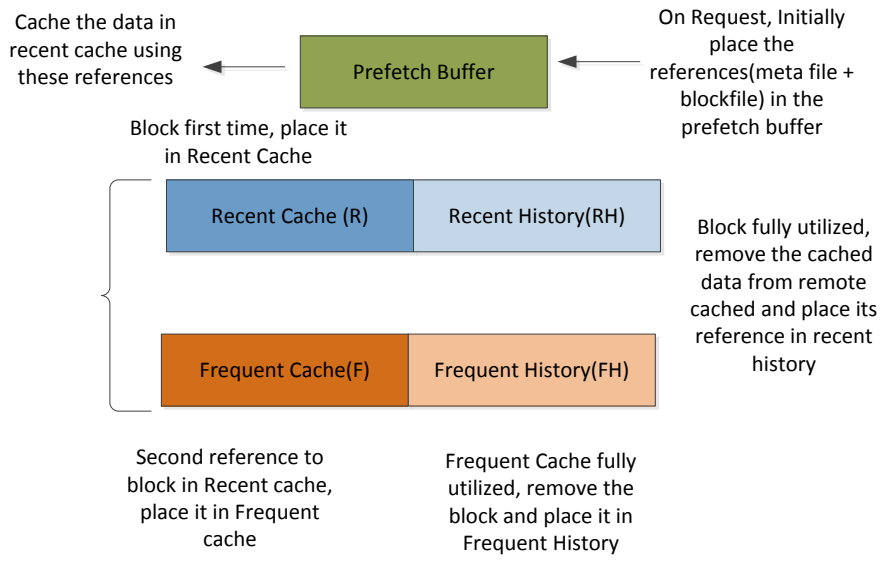

Figure 5.3: Modified-ARC

The size of recent and frequent together is fixed. The basic idea behind this is :

1. Initially on a request for block, check if the references are present in the prefetch buffer or any of the history caches. If references not found in the prefetch buffer or any of the history caches then cache them by initially placing in the prefetch buffer, if references found in either of the history caches then place the blocks in their corresponding caches(recent or frequent cache) and then blocks can be served from those caches.

2. If the references are placed in the prefetch buffer then, for the first time serve the request using these references.

3. Use these references to cache the block if not already cached and since it is the first time, place it in recent cache. If in Step 1, found in recent history cache then place

in the recent cache or else if found in the frequent history cache, then place in the frequent cache. Caching of blocks involves caching metadata for the block as well as the data for the block.

4. A next hit on the same block places the block into frequent cache.

5. When either of the caches are fully utilized then block is evicted from recent or frequent cache but its reference is placed into its corresponding history. When either of the history caches are fully utilized causes the references to just drop out of the cache.

6. A hit in either recent history or frequent history removes the reference from history and places the data in the cache. Simultaneously in the same turn, the references are served from the history as well since there is a hit in the history cache. So any hit in the history and data not in cache, initially serve from the references either in prefetch or history cache.

7. References from prefetch buffer are evicted when the prefetch buffer is fully utilized.

Advantages

- The overall cache hit rate increases
- Any hit in the history cache causes the request to be served by references.
- Prefetch buffer, in order to separate the references and the history.

Differences between new Modified-ARC and previous Modified-ARC The difference between current Modified-ARC implementation and previous proposed Modified-ARC is that there is no adaption in the current implementation since focus was to improve the MapReduce job execution and with this simple Modified-ARC implementation helped in achieving the performance needed.

5.2.3 New Caching Algorithm

1. Initially DFSClient requests NameNode for a read on a file with a defined offset or index.

2. NameNode does a lookup for the blocks that make up the file and the DataNodes where the blocks are located.

3. Since NameNode maintains the Global Cache Image and it builds up the mappings of cached block to DataNodes from the cached block report obtained, hence NameNode sends as response to DFSClient request the list of locations containing cached DataNodes locations, non cached DataNode locations, replicas and block ids.

4. DFSClients receives locations and sorts it such as the cached block locations are always first in the list and then the non cached block locations.

5. DFSClient tries to choose the best node to connect to. It parses through the list of cached deadnodes and non cached deadnodes, so that DFSClient does not attempt to connect deadnodes. Deadnodes are nodes to which DFSClient failed to connect. Also check is made if cached read attempt has been maxed out. This is done so that forwarding of the address by the DataNode to other DataNode does not lead into an infinite loop.

6. DFSClient attempts to connect to first DataNode in the list after removing the deadnodes and cached deadnodes and checking the cached read attempt.

7. It also checks if ”dataNodeFromOtherDataNode” parameter is empty. This indicates, this is the first time it is connecting and it is not asked to redirect its request, that is not connect to other DataNode.

8. If a cached block location is found and it attempts to read from it, then it sends a opcode indicating reading from cache to DataNode.

9. While DataNode is trying to serve the request from cache it checks for error. The error can be opcode of block being cached somewhere else or the opcode can be that it cannot find the cached block and try reading from the non caching DataNode.

10. When DataNode indicates that block is cached somewhere else, it returns back to DFSClient with address of the other DataNode to connect to which is collaborative caching. When DataNode indicates that block is not cached at all, then block is read from the non caching DataNode.

11. DataNode sends a periodic cached block report to indicate the blocks are present in the local cache.

12. NameNode maintains the Global Cache Image and updates it based on the cached block report obtained from DataNode. As a response NameNode sends command to update the GCI on DataNode.

13. When the request arrives at the DataXceiver of the DataNode and if opcode is read from cache then check for errors, if no errors then stream from cache. If the request is for the first time, then check if the reference for this is cached in history cache or in prefetch buffer, if cached in history then do not place them in the prefetch buffer instead use these references in history to place in the corresponding cache. If not cached in either then place the references in the prefetch buffer. Use these references in the prefetch buffer to cache the block and place it in recent cache. If the request is found in prefetch, since prefetch not fully utilized and hence the references are not evicted, then use those references to cache the block. If this request is not for the first time and is found in recent cache then serve the block and place it in frequent cache,.

14. This caching of block is done at the DataNode BlockSender end. Whenever block is to be cached and cache is full, then eviction and replacement policy is used to accommodate the new block.

5.2.4 Previous Proposed Caching Algorithm

1. Function Manager receives requests from client and checks the kind of request, if the request is other than read, open, close or rename operation then it is forwarded to NameNode. Otherwise it will check in the metadata cache, if hit then it returns the mapping of blocks of the DataNodes to the client along with the location of the replicas. Else Function Manager will retrieve a set of mappings of file to DataNodes from the namespace and forward it to clients.

2. Contacts the Cache Managers of the other DataNodes to prefetch the blocks related to the FileID and place it in the prefetch buffer.

3. Clients will contact Cache Managers of the DataNodes from the information received by the Function Manager.

4. When Cache Manager receives the request from the client, Cache Manager will check if blocks are available in prefetch buffer, then serve the request or else check the local cache, if not available then queries the FIFO queue in Distributed Cache to have the recent copy. Then it checks Remote Cache List to find which other DataNodes hold that particular block and forward request to the Cache Manager of that DataNode containing that block. If block is not cached at all then retrieve block from storage and add it to recent list of that particular DataNode.

5. Two cases arise, when Function Manager tries to find the mappings in Namespace.
   Case 1. Function Manager locates mappings in the current Namespace
   In this case follow the steps 1 to 4.
   Case 2. Function Manager does not locate the mappings in the current namespace
   The Function Manager first checks the global cache for the cached blocks and metadata. If it is a miss then Function Manager contacts the NameNode of other cluster and triggers Global Manager to cache the NameNode ID and blocks from the other cluster.

6. To maintain data consistency whenever any changes are made to the contents, a file named 'dirty file' will contain the list of block ids that have been changed and on which DataNode ID have they been changed and this file will be checked in T units to be updated. Like if a file is deleted then its blocks are deleted as well in that case the blocks are to be deleted from the cache as well. Hence an entry for the corresponding blocks will be reported in the 'dirty file'.

7. Replication factor will be reduced to 2 so that when the access frequency reaches a certain threshold then the replication factor of the blocks will be increased. This will help in saving disk space.

8. In case of eviction of singlets or moving blocks because of full cache utilization block replacement and block forwarding algorithm are used.

5.2.5 Reason for Adoption of New Algorithm

There is an overall change in the proposed algorithm mentioned in the proposal. A new architecture as well as new algorithm has been proposed. The reason for this is because primary focus is on performance. A simple new architecture with the implemented new algorithm helped in achieving high performance and lowering the MapReduce execution times hence meeting the required goal. In previous architecture, there were more lookups and could increase the overall execution time. An in depth study of the Hadoop system, code and initial experiments performed helped in proposing the new architecture. DFSClient has to always contact NameNode to get the set of locations, so the idea of NameNode sending the cached locations as well does not require extra lookup. DataNode does provide heartbeat, block report and in response NameNode asks it to perform a task. So NameNode maintaing the Global Cache Image would be a good idea where it does not require extra work because it builds up the namesystem from the block report received from DataNode. Similarly it just adds the cached locations mappings to the list of non caching

DataNode list. The lookup of cached block in case of a local cache miss is done by DataNode. The replacement policy, eviction policy are the responsiblities of the individual cache managers. Hence the overall proposed new algorithm and architecture is simple and does not incur extra load either on NameNode or DataNode.

5.3 MapReduce Job Execution [Hypothesis Proved]

- The proposal to implement collaborative caching and integrate with Hadoop works. It proved to be successful resulting in a considerable lower job execution times which led to improvement and enhancement in the overall system.
- Reading data from cache is always faster as compared to reading data from the disk. On the DataNode side, the data was being streamed from disk hence the overall performance of a MapReduce job was considerably slow and had a overall high I/O rate.
- An attempt has been made to cache this data and stream this data from cache. Moreover, collaborative caching allows us to stream the data from remote caches which proves to be an added advantage.
- In Hadoop, it was also observed that caching references improved the system performance by some amount.
- So, when collaborative caching was implemented a considerable improvement in the job execution time was observed.
- There are three main reasons that contribute to improvement in the system. First reason, when first time a request is encountered and a cache check is made for the requested, if data is not found in cache causes the data to be cached. The main advantage here is that with the initial request, only certain number of bytes are read and not the actual data. The same request for the same block comes again and this time it is already cached, so the data is streamed from cache. But with the request for the

first time, there incurs extra time to cache the block. But during the initial execution itself, we can observe some difference in timings as compared to the default configuration. The second time,*a request for the same block a considerable decrease in the timing can be seen, as the data is being streamed from cache*.

- The second reason is, whenever JobClient submits the Job, JobTracker tries to schedule the job on the same node as the input. More data-local tasks, better the execution time. So as soon as TaskTracker contacts JobTracker either with slot available or no, if the slot is available then schedule a task. If the input required for the task resides on a different node, then it is rack-local and in such a case the data is streamed from other node to the node where the task is scheduled and then the task is carried out. This results in added extra time to complete the task resulting in increasing the overall job execution time. But with caching, *the tasks scheduled get completed earlier which provides room for tasks to be scheduled as data-local, hence improving the overall execution time*.

- The third reason was, initial request served by the references cached during the request contributed to the improvement as well. This is because, these references causes the system to obtain the data into cache faster since disk lookup with large number of files incurs extra time and with reference caching, this lookup is saved. With the Modified-ARC cache replacement algorithm, better cache hit ratios were observed, a factor leading to overall improvement. Prefetching of references was another contributing to improvement.

- The overall improvement is also result of the caching algorithm. Since DFSClient always sorts caching locations as first in the list other DataNodes which shows that attempt is always made to stream from cache. Moreover, if DFSClient cannot find the block in the local cache of this DataNode, it can connect to next location provided by DataNode for streaming from cache.

5.4 Removed Components proposed in previous architecture

Following are the components which are not considered for the current architecture:

- **Additional functionality on Distributed Cache**: This functionality of maintaining an extra FIFO queue for global cache updated is not considered. The reason for this, DFSClient always contact NameNode. Also the DataNode always send block report and heartbeats so it was a good idea for NameNode to maintain the overall Global Cache Image instead of maintaining it on a Disitrbuted Cache.

- **Function Manager** This functionality to be able to divide the overall load of NameNode could not be implemented due to time constraint.

- **Global Cache Manager** This functionality of implementing remote caching between the clusters could not be implemented due to time constraint.

Chapter 6

Hadoop-CC Implementation

Chapter 5 gives us an overview of the architecture and the algorithm of the proposed design. This chapter explains in detail how different components interact with each other and the different Data Structures utilized by them. The implementation was done in Java 1.6 and integrated into existing framework of Hadoop.

6.1 Interaction among Components

6.1.1 Interaction between DFSClient and NameNode

Below Fig 6.1 shows a sequence diagram for interaction between DFSClient and NameNode

- DFSClient requests NameNode for blocks locations depending on the file and offset. NameNode receives the request and it contacts the FSNamesystem for the set of blocks which make up the file and the set of DataNodes where these blocks are located.
- Since NameNode is the central server for maintaining the global cache image, it contains the list of DataNodes where the blocks are cached.
- FSNamesystem returns the blockids and set of locations where these blocks are located along with replicated blocks. FSNamesystem also does a lookup if the blocks are cached at those DataNodes. If it finds cached blocks, then returns set of caching DataNodes as well as set of non-caching DataNodes to NameNode.

- NameNode in turn returns this list to DFSClient in the form of Located Blocks.
- On receiving the list from NameNode, DFSClient always tries to determine the best DataNode to connect. It sorts the list such that caching DataNodes are first in the list and the non-caching nodes are arranged after them.

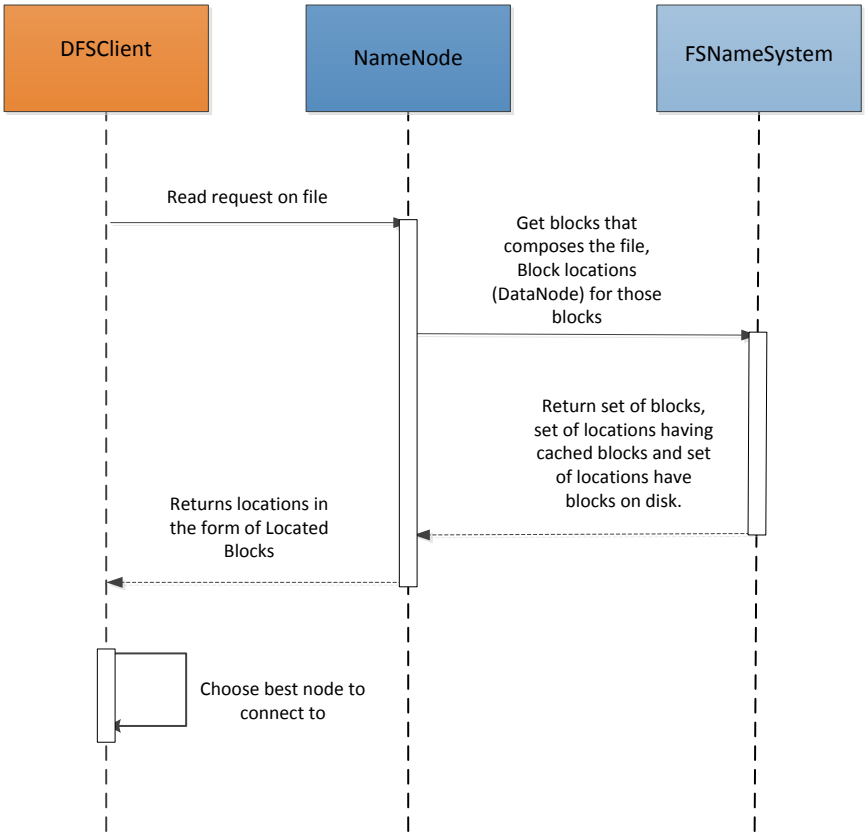

Figure 6.1: Sequence Diagram for Interaction between DFSClient and NameNode

6.1.2 Interaction between DFSClient and DataNode

Fig 6.2 shows interaction between DFSClient and DataNode. The interaction between them can be explained as follows: Above shows a sequence diagram for interaction between

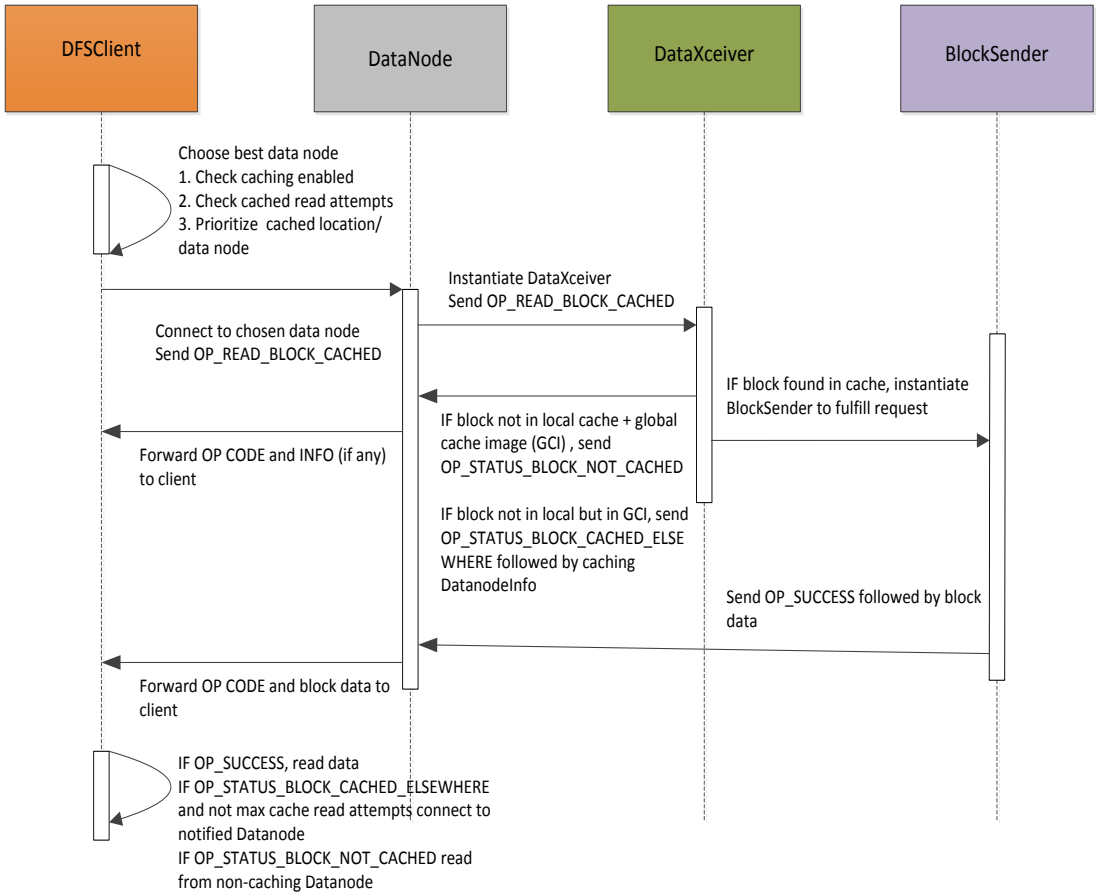

Figure 6.2: Sequence Diagram for Interaction between DFSClient and DataNode

DFSClient and DataNode

- After obtaining locations of the blocks and their block ids from NameNode. DFSClient sorts them in the order such that caching DataNodes are first in the list. It checks for total cached read attempts. If the cached attempts has crossed a certain limit then connect directly to non caching DataNode.

- It checks if the DataNode it is trying to connect was previously declared dead. A

dead DataNode is a node when the client tried to read earlier and after two attempts, the DFSClient failed to connect to DataNode. Similar to deadnodes are cached deadnodes which indicates that particular DataNode previously had it cached, cached block was removed and the block could not be obtained. It sorts and tries to connect to best DataNode possible.

- A connection is established between the first chosen DataNode in the list and DFSClient. If DFSClient is reading the cached block then it writes opcode OP_READ_BLOCK_CACHED to the DataNode. DataNode in turn instantiates DataXceiver and forwards the opcode sent by DFSClient.

- If cached block is found then opcode OP_SUCESS is sent followed by data using BlockSender and is forwarded to DFSClient so that it can start reading data.

- If the cached block is not found then send OP_STATUS_BLOCK_NOT_CACHED opcode or OP_STATUS_BLOCK_CACHED_ELSEWHERE to DFSClient. Opcode OP_STATUS_BLOCK_NOT_CACHED indicates no cached block can be found and opcode OP_STATUS_BLOCK_CACHED_ELSEWHERE indicates this block can be obtained in remote cache. This opcode is followed by the socket address of the DataNode to connect to.

- After receiving the opcode OP_STATUS_BLOCK_CACHED_ELSEWHERE, it again checks for cache read attempts, if not maxed out then try reading from cache specified by other DataNode, if caching DataNode available.

- If opcode OP_READ, then read the data from non caching DataNode.

- If the read is for the first time, DataNode indicates CacheManager to cache the block's metadata as well data.

6.1.3 Interaction between NameNode and DataNode

Above Fig 6.3 shows interaction between NameNode and DataNode

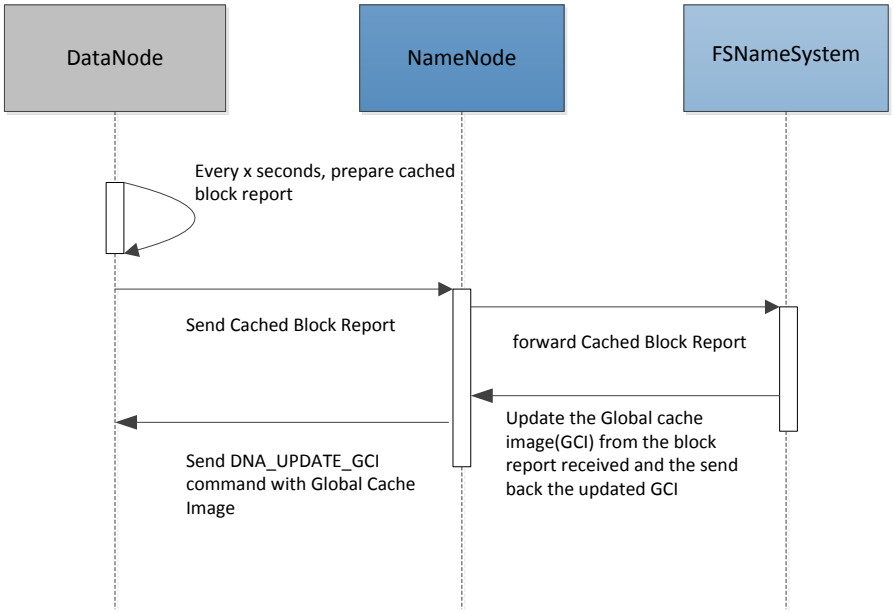

Figure 6.3: Sequence Diagram for Interaction between NameNode and DataNode

- DataNode sends NameNode cached block report about its locally cached blocks after a certain configured interval. This is sent in the form of CachedBlocksCommand
- Accordingly NameNode forwards the request to FSNamesystem and updates the Global Cache Image in the namesystem as it receives the report.
- As a response to this command NameNode sends DataNode instruction to updates its Global Cache Image with the help of DNA_UPDATE_GCI command.

6.2 Implemented DataStructures

The different datastructures used during the implementation are as follows:

CachedBlocksCommand private Map<Block, Set<DatanodeInfo>> blockMap.
Each of the CachedBlocksCommand is a mapping of block to many DataNodeInfo objects. DataNodeInfo is an object which maintains the status of DataNode like dfs capacity, dfs used, storage and caching enabled or disabled.

Caching

- Recent/Frequent cache:
  private final Cache recent;
  A recent cache is composed of block and CachedBlock object. Each CachedBlock contains block(bytes, generation_timestamp,id), metadata and data.

- CacheBlockMap in FSNamesystem:
  final Map<Block, Set< DatanodeInfo>> cachedBlocksMap =
  Collections.synchronizedMap(new HashMap<Block, Set<DatanodeInfo>>());
  This is mapping maintained by FSNamesystem for Global Cache Image.

- Local GCI:
  private Map<Block, Set<DatanodeInfo>> gci =
  new HashMap<Block, Set<DatanodeInfo>>();
  Each DataNodeInfo object represents the status of DataNode and is the main medium of communication between DFSClient and DataNode. It represents the status like if the block is cached to help DFSClient determine which DataNode to connect.

6.3 Impact on Overall System

- Integration of collaborative caching code into the existing Hadoop system led to overall improvement as discussed in the section below.

- MapReduce job execution times was lowered which was measured and verified by conducting experiments on varying block sizes.

- Although NameNode has been made the central server for maintaining the Global Cache Image, since it is just the references which is transmitted over the network, the data is not large. NameNode updation of the global cache image does not take time since it is just an addition to already maintained mapping by NameNode.

- Due to collaborative caching, more data-local tasks are being scheduled. The reason behind this is, when JobClient submits a job, JobTracker puts that job in the queue. This job is then picked up JobScheduler. Jobtracker on receiving heartbeat from TaskTracker where TaskTracker indicates if it has an open slot, task is assigned. JobTracker tries to schedule jobs to TaskTracker such that it is data local, that is task is executing on the same machine as the input. Otherwise if task is scheduled on a different machine then the input, it is called rack-local task and in such a case, input has to be streamed into process on the input which in turn consumes time. Also the tasks stream and read the data from disk which is I/O intensive.

6.4 Improvement

Due to collaborative caching scheme, following are the improvements observed in the overall system.

- Improvements in the execution times of MapReduce was seen. More improved results for caching can be obtained with faster network speed roughly around 1 Gbps.

- Main improvement was due to collaborative caching introduced, data-local tasks and reference caching.

- Improvement was resulted due to DFSClient having the information about DataNodes which have these blocks cached so that attempt is made to connect to caching DataNode.

- Other improvement was seen due to replacement algorithm to replace the cached data.

- In normal circumstances, we would need more nodes in order for MapReduce to execute faster. But with caching this would not be needed and with limited nodes, this improvement can be achieved.
- With collaborative caching, more MapReduce number of jobs with larger files can be performed at lower rates. This is needed due to fact of ever increasing rate of data.
- Multiple client job execution times were seen to overall improve.

Chapter 7

Comparison

This chapter explains the differences in approach mentioned in the related work with respect to the proposed algorithm of Hadoop-CC.

7.1 Related Work vs My Contribution

7.1.1 Dynamic Caching using Memcached

- In case of Dynamic Caching using Memcached, Memcached servers are centrally coordinated. To read a file, request is simultaneously forwarded to both Memcached and NameNode and both of them will return set of locations. If memcache returns true then block is cached and that location is considered for read. In second architecture discussed in the paper, DataNode and NameNode waits for MemCached to return with a reply, and if cached data is not found then NameNode replies to the request to the DataNode. In both the architectures, we are dependent on Memcached for the reply, also second architecture incurs an added delay if Memcached does not contain the cached blocks.

- In Hadoop-CC approach, we get the locations from NameNode which contains cached locations nodes as well as non cached locations. The sorting of this is based on priority of reading from cached locations on the client side. In this there is no waiting time as compared to the waiting time observed in the Memcached.

- The other difference is prefetching is done for references which shows the improvement in the first run itself. While for Memcached, the prefetching is done once request arrives.

- The next is the replacement policy followed is LRU. But in Hadoop-CC the replacement policy of Modified-ARC.

- There is no remote cache lookup, all the lookups are done by the Memcached. In my approach if there is a local cache miss on a DataNode, then that DataNode asks client to connect to other DataNode and the information is provided by the DataNode itself. By this policy lookup on NameNode is saved.

7.1.2 PACMan

- In PACMan, the cache replacement policies are applied at the coordinator. There is no policy of remote cache in PACMan, If not found in local cache then just serve from disk.

- Instead in Hadoop-CC approach, if the request cannot be served from local cache, then a lookup for remote cache is done. For that we do not contact the NameNode, instead we do a lookup at the DataNode in the Global Cache Image with the help of CacheManager.

- They have individual PACManClient components which are installed on the DataNodes which manages the caching of the blocks. They use sticky policy[7], for improving the average completion time they use LIFE, for cache replacement policy they use LFU-F. They try to schedule tasks such that they are data local. The algorithm also takes care of the incomplete files and calculation the wave-width[7].

- In Hadoop-CC approach we get the cached locations from NameNode, sorting is done on the DFSClient to connect to the best DataNode, remote caching is enabled

in case of local cache miss. Modified-ARC algorithm used for replacement policy. More data-local tasks are scheduled due to caching.

Chapter 8

Initial Analysis

In order to understand the working of hadoop system and its behaviour a series of initial experiments were performed.

8.1 MapReduce Job

8.1.1 Psuedo Distributed Experiment

In order to understand the Hadoop default system, my first experiment was running MapReduce job on a set of data files. The MapReduce job executed was WordCount. All the files were test files. The data files ranged from 10 MB-500 MB in increments of 10 and the block sizes in the increments of 4 MB. This experiment was performed on a pseudo-distributed environment.

Observed Results The timing was recorded using the inbuilt Linux ”time” function. It was observed that total processing time to complete a MapReduce job was on
”avg execution time” : 300 secs as compared to
system time 0.40 sec

Conclusion This observation led to conclusion that 90% of the time, the system is waiting which is thread is in the blocked state for a long time.

8.1.2 Cluster Environment

When experiment was conducted in a fully distributed environment similar results were obtained. avg execution time" : 500 secs as compared to
system time 0.54 sec

Conclusion In a MapReduce task, after the completion of map phase, reduce phase does not start immediately but it waits, [based on 7]. It was observed that there are more Rack-local jobs scheduled. The streaming of data is from the disk.

8.2 Reference Caching

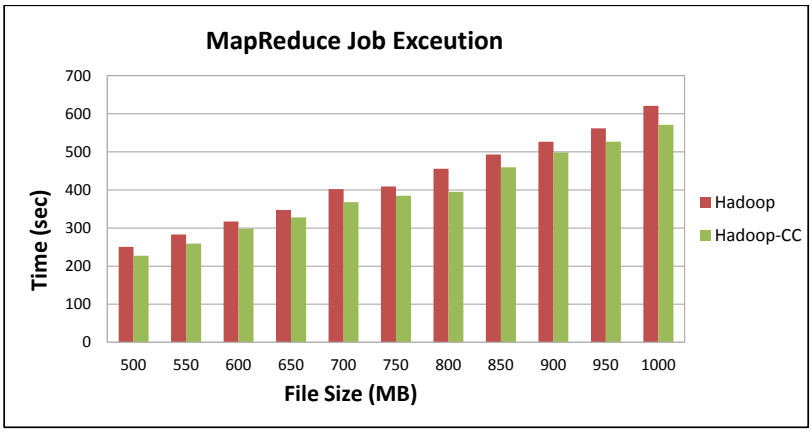

Figure 8.1: Reference Caching (Block Size 64 MB)

Experiment carried out on Psuedo-Distributed The next experiment performed was placing the file references in the cache. A Block is composed of meta file and block file. Metafile is also known as the checksum file. Those file references were placed in the cache. This experiment was performed over the data files 10 MB-500 MB for pseudo distributed environment in increments of 10 and for clustered environment, file size 500 MB- 1 GB in increments of 50. Fig 8.1 is a graph showing MapReduce Exceution times for block size

64 MB for reference caching on cluster environment. Fig 8.2 shows a graph of MapReduce Execution for block size 64MB on psuedo distributed environment.

Observed Behaviour For a psuedo- distributed environment,it was observed that for small files upto size 380 MB, there was just a slight difference in the readings of Hadoop and Hadoop-CC. As the data file size increased, the difference in the execution times could be observed. An interesting observation was that, even though the file size difference between 220 and 230 is not much but the timing difference was observed such that timing of 220MB file is 138 secs while that of 230 MB file size is 132 secs. The reason for this MapReduce internally has other phases such as for map - map and sort while for reduce - shuffle, sort, combine. If we see figure 8.3 and 8.4, a higher value was observed for parameters SLOT_REDUCE_MILLIS Mapped input records, Reduce shuffle bytes for file size 220 MB as compared to 230 MB. Even though file difference is that of 10 MB, but it depends on the amount of work the different phases in map or reduce has to do, to carry out tasks on data. After the file size of 400 MB, improvement was seen in the overall execution time. Only after 380 MB file size, difference in the timings of MapReduce execution times on default and that of Hadoop-CC can be observed. For MapReduce Execution times on clustered environment for block size of 64 MB, difference in execution times of Hadoop and Hadoop-CC is observed with lower execution times for Hadoop-CC.

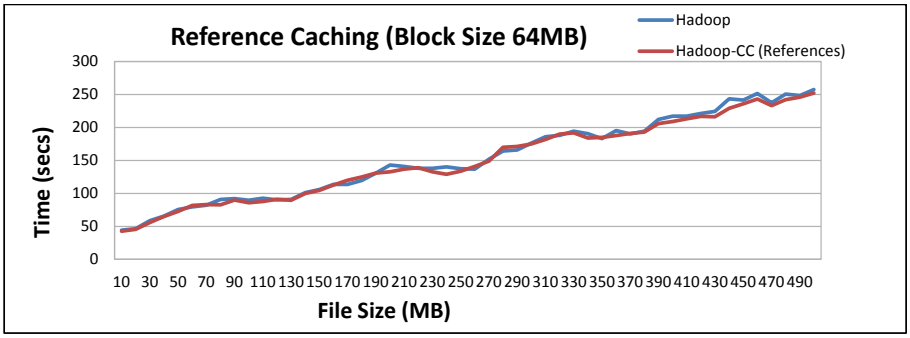

Figure 8.2: Reference Caching on Pseudo Distributed (Block Size 64 MB)

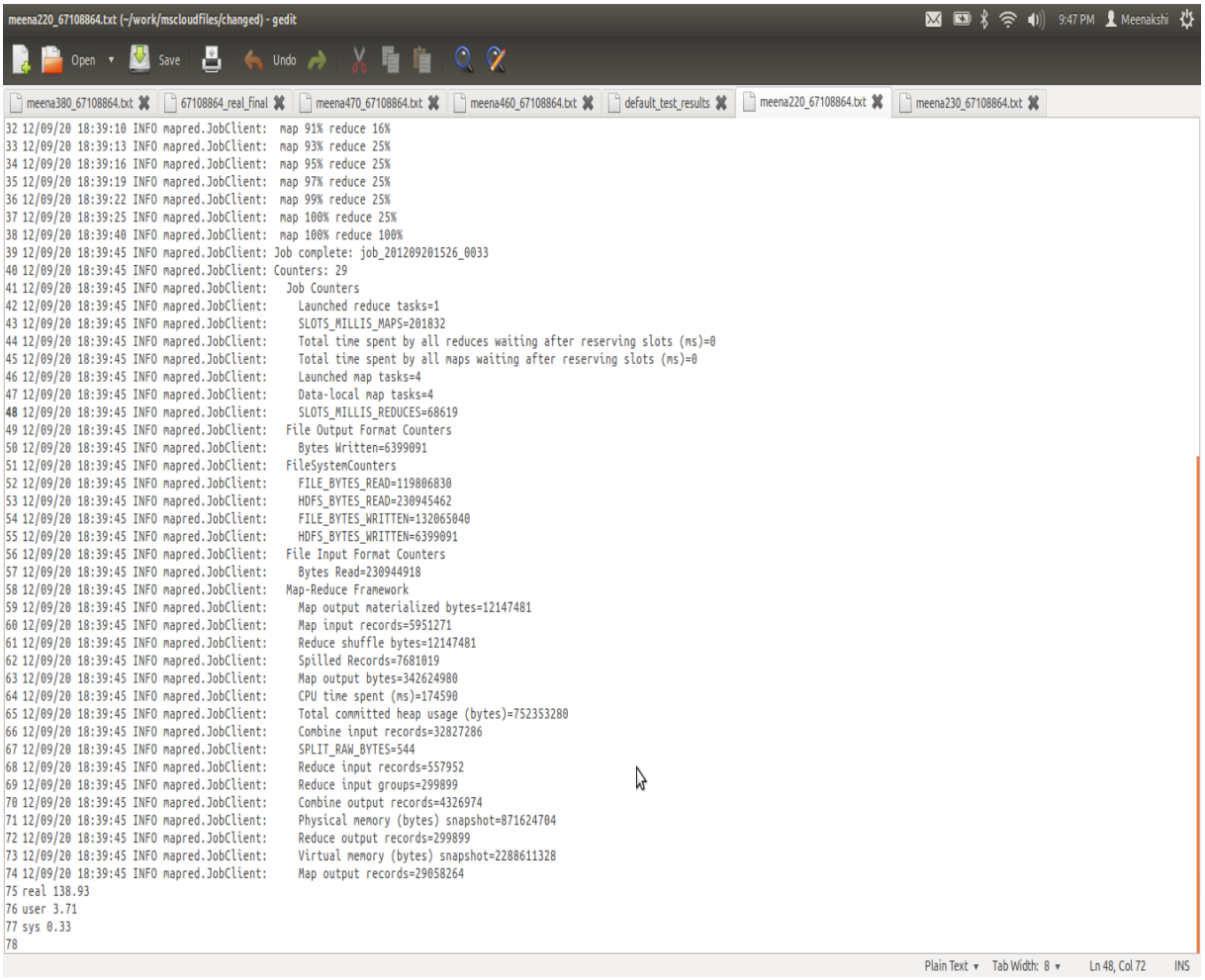

Figure 8.3: Reference Caching 220 MB File Size (Block Size 64 MB)

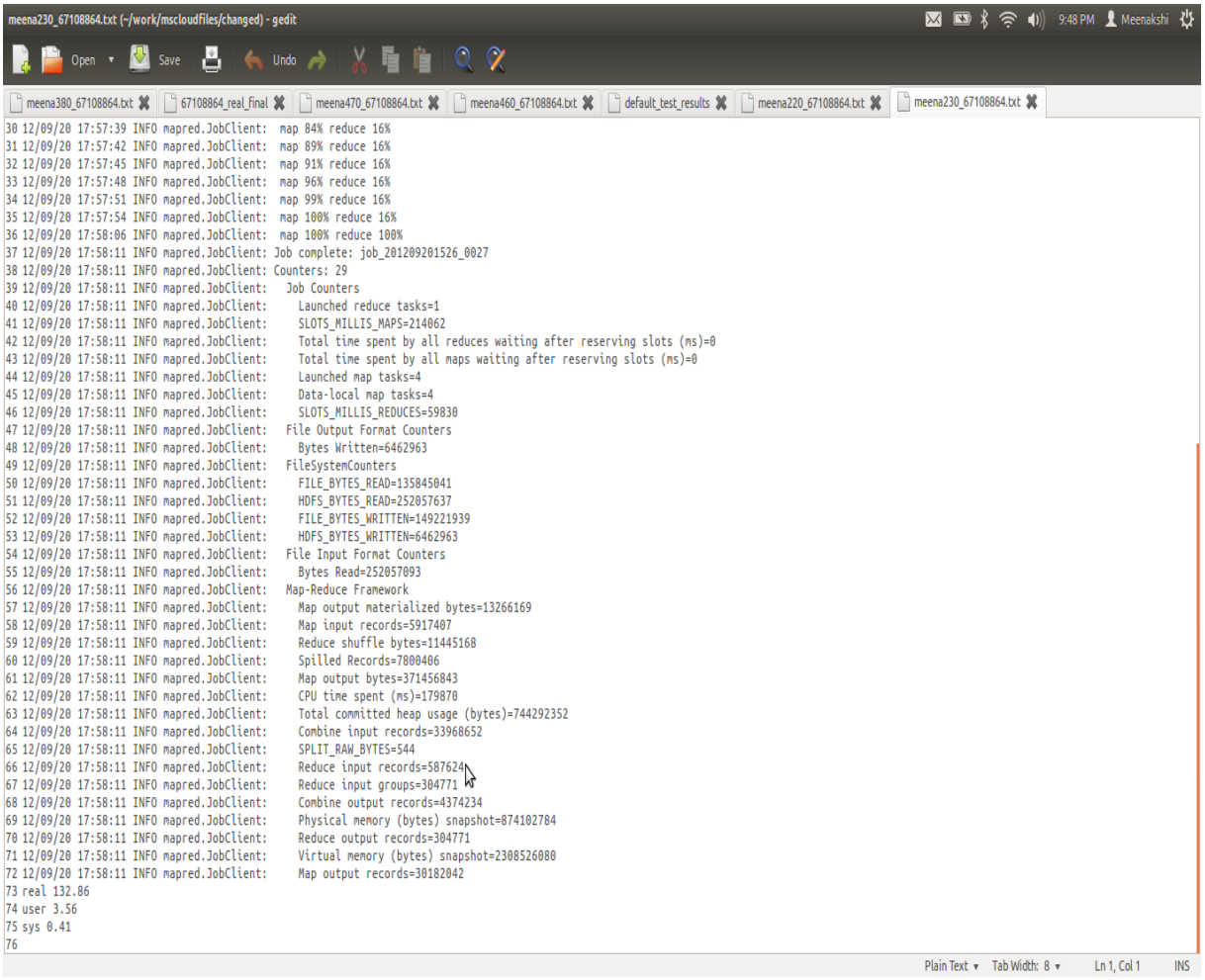

Figure 8.4: Reference Caching 230 MB File Size(Block Size 64 MB)

Conclusion Caching references increases the overall execution time because before starting the streaming of data, it looks up for those files on disk and that consumes time. By placing in cache, the lookup time reduces and hence the access time reduces.

8.3 DFSClient Caching

Experiment An effort was made to see if caching can be done on the client side. For this experiment, a track was kept about the DFSClient id and the instance count.

Observation It was observed that we always had a new DFSClient ID for the same session. Also the instance count did not increase and was constant at 1. When two jobs were submitted parallely each of them showed the individual count as 1 and their DFSClient ID was different as well.

Conclusion The reason for this is each time when a request is made, a new JVM is launched. When the job finishes, the connection is closed and JVM exits. Hence there is no way of tracking the DFSClient ID.

8.4 Further Caching Improvements

Further improvements can be obtained in terms of Hadoop-CC by having network speed of 1 Gbps

Chapter 9

Evaluation

To study the behavior of default configuration of Hadoop and evaluate Hadoop-CC's performance, a cluster was setup with limited nodes. The results obtained from Hadoop-CC were compared with default configuration of Hadoop.

9.1 Experimental Cluster Setup

The cluster consisted of a single NameNode and three DataNodes and all the three DataNodes were placed in the same Rack.

9.1.1 Hardware Configuration

Following was the hardware configuration of all the nodes in the cluster.

- All Machines had processors Dual-Core AMD Opetron
- Among those, three machines had memory capacity of 2GB while one machine had a capacity of 1GB.
- The network speed available was 10/100 Mbps.
- Storage capacity available was 200 GB on one of the machines and 160 GB on the remaining three machines.

9.1.2 Software Configuration

Following softwares were installed on all the nodes in the cluster.

- Ubuntu distribution 10.04 (Lucid Lynx).
- Java version 1.6 (SUN)
- SSH server was installed for seamless login between the machines.
- Hadoop distribution 1.02 [17]

9.1.3 Experiments

Experiments were performed on default configuration of Hadoop and compared with results obtained on execution of jobs on Hadoop-CC. The replication factor was reduced and set to 1 due to storage limitations. Due to memory limitations and to measure the effective performance of collaborative caching, collaborative caching experiments were conducted on a smaller datasets.

Job Execution For all the experiments, the WordCount problem was chosen as the job to be run. The WordCount job calculates the frequency of all the words in the given input file(s).

Common configurable parameters are

- Local Cache Size: To calculate the cache hit and miss rates, recent cache size and frequent cache capacity, parameters were configured from 6 - 24 depending on the block size.
- Block Size: To study the initial behavior on pseudo distributed, the block size was varied from 4 MB 128 MB. To perform experiments on cluster, the block sizes used were 16 MB, 32 MB, 64 MB, 128 MB.

- Minimum Block Size For Caching: The blocks having size of greater than 1 MB were considered for caching.
- History Cache Size: This size was configured to 100.
- Buffer Size: A buffer size of 32 KB was set to cache the data.
- Max Read Cached Attempts: This number is set to 2.
- Caching Enable: To enable or disable caching on Hadoop.
- Cache Block Report Interval: This parameter is configured as 10 secs.
- Prefetch Buffer size: The configured prefetch limit is 100.

9.2 Assumptions

- Cache is read only.
- Cache size on all the DataNodes are configured of same size.
- All the failures are not taken into consideration.

9.3 Metrics

- Average Block Access Time:
  Considering large block size of Hadoop which is by default 64 MB, Hadoop does not read all the data at once, instead it streams the data in the form of packets and then sends over the network to recipient. Hence for Hadoop the total average block access time is the time taken to read the data and time to transmit the data over the network. Total Average Block Access Time = Time to read from disk or cache + Time to transmit the data over the network.

- Cache Hit / Miss Rate:

  It is used to measure how many times a block was read from the cache. The higher the number, the better it is.

  Cache hit rate is calculated :

  Hit Rate (%) : Total no. hits / Total number of requests for blocks.

  Total Hits (%) : Total no. hits in local cache(recent + frequent) and global cache

- Local Cache Size:

  Local Cache size is a combination of Recent cache size and Frequent cache size.

  Local Cache Size : Recent Cache Size + Frequent Cache Size.

  So if we say that Local Cache size is 18 it means Recent Cache Size is 9 and Frequent Cache Size is 9.

- Max Read Cached Attempts:

  This number indicates the maximum number of times we connect to DataNode to read from cache before we finally read from the disk. This metric is important otherwise there will be an endless loop in which one DataNode forwards the read to other DataNode for reading the data from cache and the loop will never end.

- Cache Block Report:

  It is a report send by all the DataNodes regarding information about their local caches to NameNode. This helps NameNode construct the Global Cache Image.

9.4 Data Preparation

The dataset prepared consisted of data files in the increments of 10 from 10MB to 500MB and in increments of 50 from 500MB to 1GB. All the files were text files. The data for the files was taken from the site guttenberg[14]. The first dataset was used in a pseudo-distributed environment to study the behavior of the system. The second dataset was used

File Size(MB)	Hadoop	Hadoop-CC
500	250.69	129.34
550	283.11	144.04
600	317.35	180.28
650	347.70	238.23
700	401.97	247.84
750	408.96	214.65
800	455.72	267.89
850	493.15	286.12
900	526.50	348.56
950	561.75	325.67
1000	620.45	360.39

Table 9.1: MapReduce Job Execution (Block Size 64MB)

to study the cluster behavior and to perform experiments on default configuration as well the Hadoop-CC.

9.5 Graph Results and Discussion

Considering block size as 64MB, as the file size increased, the number of blocks were increased too. Due to memory limitations, it was difficult to hold all the blocks in the memory. Hence in order to take the readings,the cluster was restarted for dataset 750-1GB after every *dataset* like after running for dataset 750, the cluster was restarted when job was executed for dataset 800. For data files ranging from 350MB-650MB there was no need of cluster restart. So as to calculate the cache hit ratios, dataset of range 300MB to 650MB was considered for 64MB block size and 150MB to 650MB for 32MB block size.

9.5.1 MapReduce

Experiment Conducted For MapReduce experiments, after deploying the changed code, the MapReduce Job was run on datasets ranging from 500 MB - 1 GB on default as well Hadoop-CC in cluster mode. It was conducted for block sizes 32MB, 64MB and 128 MB.

Inbuilt *"time"* command of linux was used to compare the results. Cluster was restarted for Hadoop-CC when MapReduce Job was run for data files greater than 650MB, due to memory limitations. Fig 9.1 and 9.2 are graphs for MapReduce Job Execution for block size 64MB. Fig 9.3 and 9.4 are MapReduce Job for block size 32 MB and 128 MB respectively. Table 9.1 shows readings for MapReduce job run for 64MB block size.

The results varies depending on the distribution of blocks across the cluster and the way the jobs are scheduled across the cluster An average of 5 runs was taken for results. Since results varies depending on the distribution of the blocks, the copying of files was carried out atleast thrice and results were obtained.

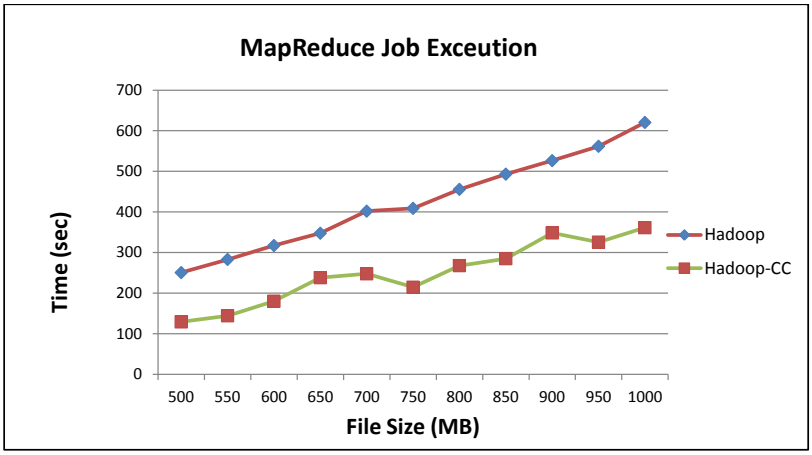

Figure 9.1: MapReduce Job Execution Line Graph (Block Size 64 MB)

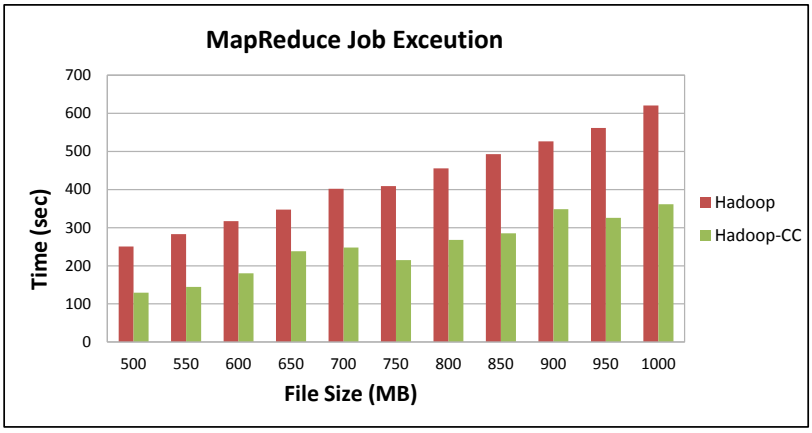

Figure 9.2: MapReduce Job Execution Bar Graph (Block Size 64 MB)

Observations It can be observed that Hadoop-CC results obtained showed a considerable improvement. As seen above, three different graphs can be observed which are for block size 32MB, 64MB and 128MB and for block size 64 MB two different types of graph which is line and bar graph. Bar graph shows the improvement, and line graph shows in detail curves.

Results It can be seen that with increase in file size, the map reduce timings have decreased. It can be observed from the 64MB graph that there are drop in timings when file size id 750 MB and 950 MB. Also it can be observed that with increase in file size, difference in the job execution time of default Hadoop and Hadoop-CC increases. A similar trend is observed for MapReduce job execution of block sizes 32 MB and 64 MB. Hence collaborative caching shows a significant improvement in MapReduce job execution times.

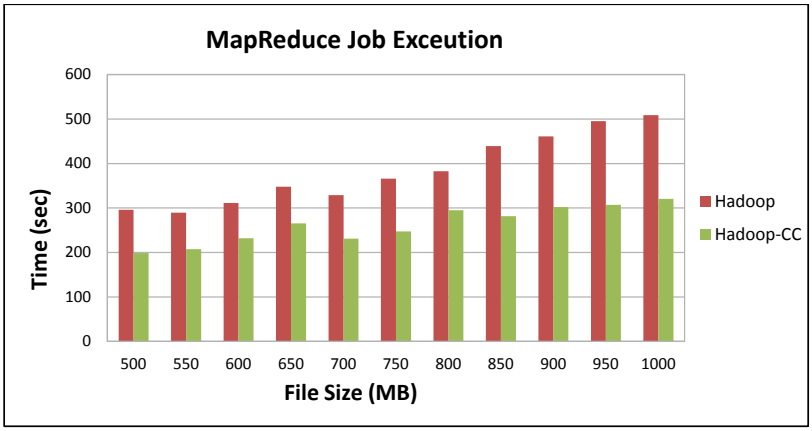

Figure 9.3: MapReduce Job Execution Bar Graph (Block Size 32 MB)

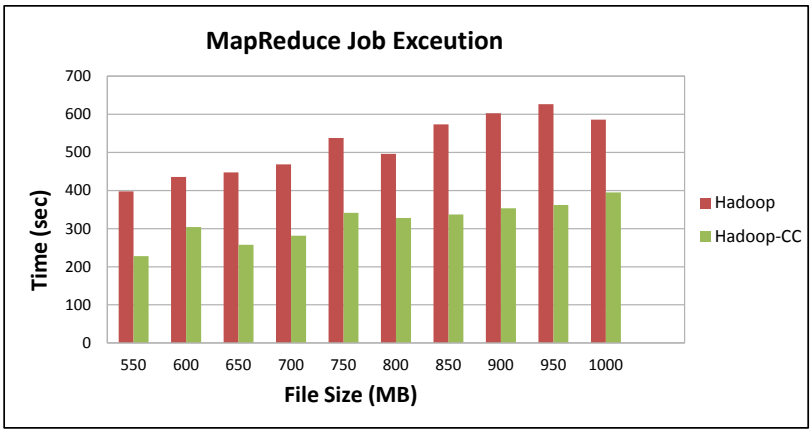

Figure 9.4: MapReduce Job Execution Bar Graph (Block Size 128 MB)

Expalanation This is due to fact that streaming from cache is faster as from disk. Also effort is been made to serve from references. As the file size increases, the no of blocks that make up the file increases. If those blocks are found in cache the timing decreases linearly. For 500 MB 8 blocks make up a file and we find those blocks in cache, we see a decrease in timing of overall MapReduce task. But if 1GB file is considered, we find that 16 blocks make up the file and we find those blocks in cache, this will lead to a greater timing decrease. Hence as the file size increases, no . of blocks increases and when these blocks are cached, it results in a higher decrease in the timings resulting in a larger gap between the job executions of Hadoop and Hadoop-CC. The other reason for the decrease

in the timings is jobs being executed as Data-Local. Jobs can be executed as Data-Local or Rack-Local. When jobs are executed as Data-Local the input lies on the same node as the job is scheduled hence its execution is faster as compared to Rack-Local when the job is scheduled on a different node and input resides on a different node. In such a case, the node at which the job is executed streams the data from other node where the job is scheduled. The more Rack-local jobs are executed, the more the time required to complete the overall job. There are two reasons when there are Rack-local jobs executed. As number of slots are fixed, for slots are being used by TaskTrackers, if JobTracker receives a request from TaskTracker and it has an empty slot, a task is scheduled, an attempt is been made to schedule a data-local task. But if that is not possible then Rack-local task is scheduled. In case of caching because of the raw data in memory, jobs are executed and completed earlier and hence slots become available earlier leading to more of data-local tasks. Another advantage is in the first run itself the caching effect is seen to be utilized, because initially 516 bytes are read the whole data. Data is cached when initial bytes are read and when attempt made to read the whole data, the block is cached. But the initial run incurs some extra time for caching of data. A drop in timings from file size 900 MB to 950 MB, which is due to fact the way jobs are scheduled and the data distribution of the blocks when the input files are loaded into HDFS. *With larger block size, less map tasks are scheduled and each map task gets more work to complete. But with very small block size, more number of map tasks are scheduled and each map gets less data to work to do*.

There seemed to be overall high n/w I/O so caching results can be improved to a larger extent with higher network speed of 1 Gbps

With increase in file size, the MapReduce job execution time decreases More data-local tasks lead to better execution time

9.5.2 Average Block Access Time

Experiment MapReduce jobs were run, to observe the time required to access a block from disk and cache. The results were collected with the help of trace file. The experiment

was conducted for Block sizes 16 MB, 32 MB, 64 MB, 128 MB. There are two diagrams for average Block access time of different graph types to show the difference in the results obtained. The line graph is used for further analysis. Figure 9.5 shows the graph for average block access time and the table 9.2 shows the readings for the same.

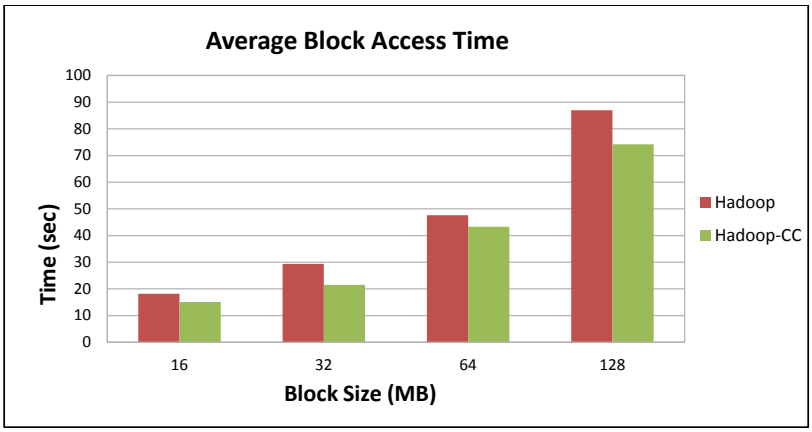

Figure 9.5: Average Block Access Time Bar Graph

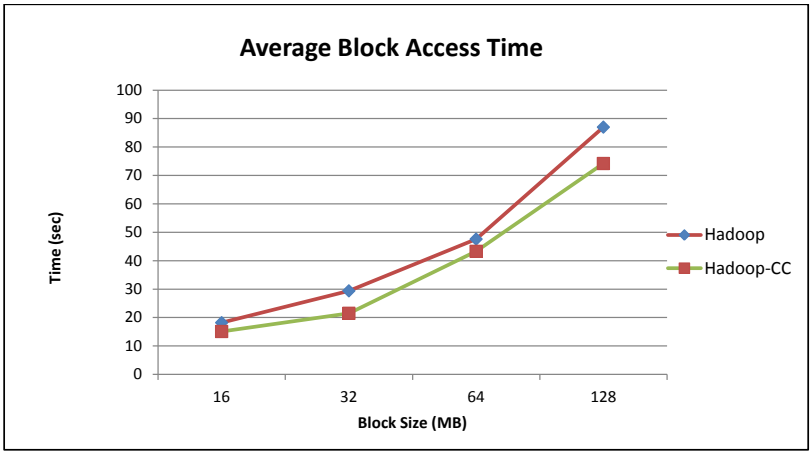

Figure 9.6: Average Block Access Time Line Graph

Observations With the increase in block size, the average block access time increases. But not much difference is observed in the average block access time of Hadoop and Hadoop-CC.

Block Size(MB)	Hadoop	Hadoop-CC
16	18.2	15.1
32	29.4	21.5
64	47.6	43.3
128	87	74.2

Table 9.2: Average Block Access Time

Explanation The reason for this is the Average block Access time here is a sum of ”time to read from disk and the time to send the data over the network” for default Hadoop, and time to read from cache and send data over the network” for Hadoop-CC. Because for Hadoop, the whole data of 64 MB is not read at once. It reads data in the form of chunks composed of packets. Hence it streams data and sends over the data to the recipient. We can see that overall average block access time does not show a significant difference. The reason for that is the n/w I/O value is high. This n/w value is highly variable and it was restricting the rate at which data can be streamed from cache.

Results A variable difference between the average block access time is observed. The variation is observed due to the n/w I/O.

Cache Capacity(No. of Blocks)	LRU	Modified ARC
6	0.12	0.15
8	0.25	0.27
10	0.32	0.36
12	0.37	0.43
14	0.44	0.5
16	0.52	0.57
18	0.63	0.69
20	0.75	0.82
22	0.85	0.87
24	0.97	0.98

Table 9.3: Cache Hit Ratio (Block Size 32MB)

9.5.3 Cache Miss Rates/ Cache Hit Rate

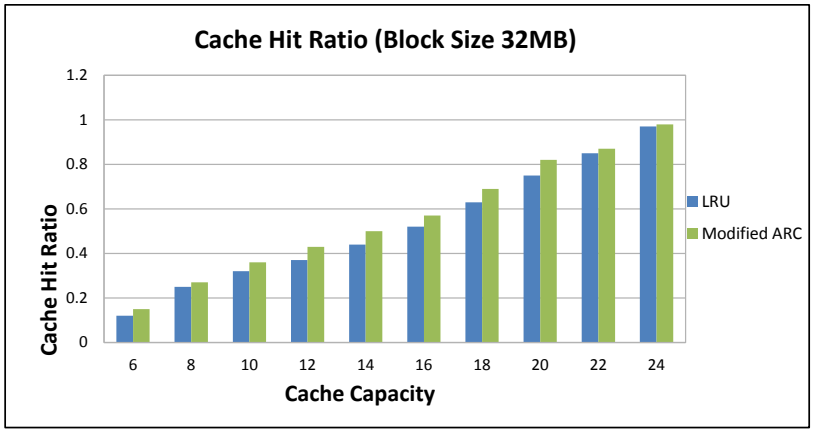

Figure 9.7: Cache Hit Ratio (Block Size 32 MB)

Experiment For carrying out experiments to calculate the cache hit rates, MapReduce jobs were run on a smaller datasets ranging from 300 MB to 650 MB for 64 MB block size and 150MB to 650MB for 32 MB block size due to memory limitations. A combination of files between this range was used to obtain the cache hit rates for LRU and Modified-ARC.

Cache Capacity(No. of Blocks)	LRU	Modified ARC
6	0.15	0.16
8	0.18	0.2
10	0.29	0.33
12	0.37	0.42
14	0.58	0.63
16	0.75	0.77
18	0.87	0.88

Table 9.4: Cache Hit Ratio (Block Size 64MB)

The no. of blocks in the cache size was varied between 6-18 for 64 MB block size and 6-24 for 32 MB block size.Fig 9.6 and Fig 9.7 are Cache Hit Ratio graphs for block size 32 MB and 64MB. Table 9.3 shows the readings for block size 32 MB and table 9.4 shows readings for block size 64 MB.

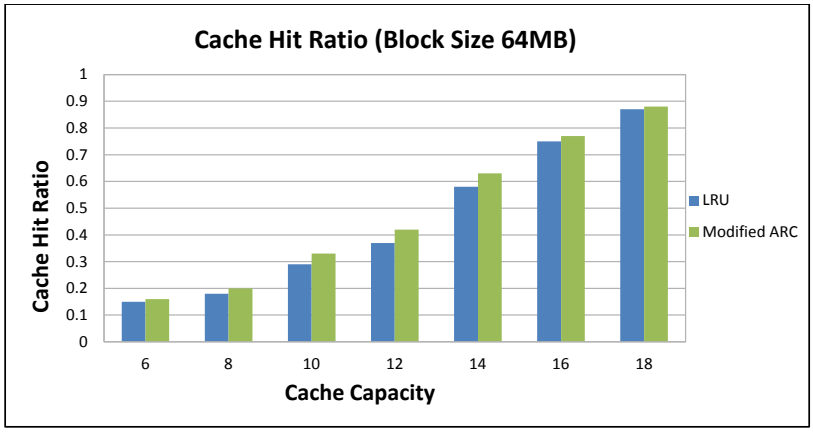

Figure 9.8: Cache Hit Ratio (Block Size 64 MB)

Configuration for Experiment

- Blocks of size lesser than 1MB were discarded.
- Experiments were performed on smaller datasets to calculate the cache hit and miss

rate due to memory limitations

When the block size was 64 MB, the blocks in the cache can only be limited to 18. This is due to the fact that Memory size of DataNodes can be given as 1.5+1.5+0.5 = 3.5 GB = 3.5*1024/64=56, which tell us that overall 18 blocks can be maintained as the local cache on each of the machines. Hence limit of 18 blocks is maintained for cache when block size is 64 MB. For block size 64 MB, it can be observed that the memory cache hit rates are not even 1.0 because the cache size is not large enough to hold all the blocks. On other hand it can be observed that when block size was reduced to 32MB. The total local cache size becomes 24 and hence the cache hit ratio increases. It is observed that when Blocks size is 64MB, the cache hit ratio is less that 1.0, but when block size is 32 MB, the cache hit raio is almost 1.0 This is because when 64MB, due to memory limitation, we cannot go beyond a certain no.of blocks due to memory limitations. But when the blocks size is reduced to 32MB, more blocks can be accommodated, hence higher hit ratio.

Observation It can be observed that with Modified-ARC, the cache hit ratio is higher as compared to LRU. It can also be observed that when block size is 64 MB, the cache hit ratio is not even 1.0, but when the block size is 32 MB, the block size can be observed to reach 1.0. It can also be observed that initially there is not much difference between the cache hit ratio of LRU and Modified-ARC. It is also observed that with larger cache size, there is just slight difference between the cache hit ratios of LRU and Modified-ARC.

Explanation A cache hit rate is computed as total hits/number of requests. A higher cache hit rate helps in better performance of the system because more data found in cache, hence faster execution time . A higher cache hit rate is observed with Modfified-ARC due to the fact we track references and cached data. With respect to cache hit ratio not 1.0 for block size 64MB, due to the fact when block size is 32 MB, the cache size is large enough to hold all the blocks, where as when the block size is 64 MB, due to memory limitations, the memory size is not large enough to hold all the blocks in cache. Since

the memory was not large to hold all the blocks as a results there were evictions. Initially there is not much difference in the cache hit ratio because with small cache size, there are more replacements, but even for smaller cache size Modified-ARC cache hit ratio is higher due to the fact of design and functioning of Modified-ARC of tracking references and frequent+ recent cache. As discussed earlier, caching references helped improve the system performance. So in effect we do not loose the block right away, we have it either in history or in cache. But for LRU, even when frequently accessed elements are placed to start of cache, it does not take into consideration frequency so in case of high load there are eventually more misses then hits. But Modified-ARC does take into consideration the frequency, it maintains history. As the cache size is increased, cache hit ratios almost become similar due to the fact that, with larger cache size, almost all the blocks are observed in the cache, hence there is no need for replacement of blocks.

Results The above graph results prove that Modified-ARC algorithm is better as compared to LRU and gives better cache hit ratio resulting in performance improvement in the system

No.Of Concurrent Jobs	Hadoop	Hadoop-CC
3	562	247.07
4	648.77	372.13
5	813.18	372.42
6	1137.0	462.58

Table 9.5: Multiple MapReduce Job Execution (File Size 500 MB)

9.5.4 Multiple Clients Execution

Experiment This experiment was conducted to run jobs parallely such that each of them are run in background. The maximum jobs which could be run was 6 due to memory limitations. Fig 9.8 and 9.9 shows bar and line graphs for Multiple Job Execution time for multiple clients. This experiment was carried out for file size 500 MB, where multiple clients are trying to run MapReduce task for file size 500 MB. Only 6 jobs could be run in parallel due to memory limitations.

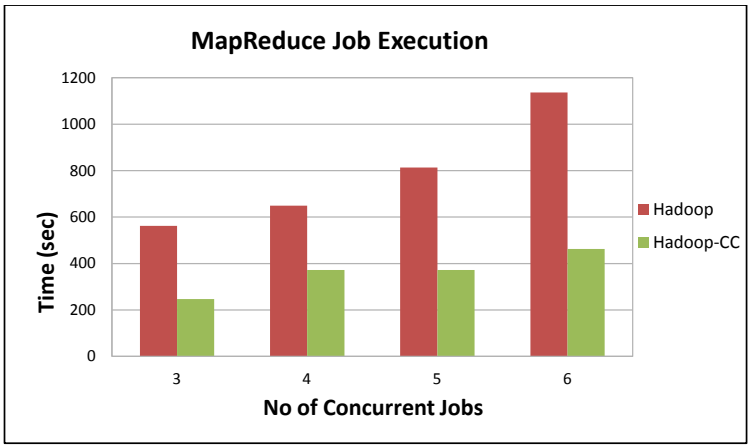

Figure 9.9: Multiple Job Execution Bar Graph(File Size 500 MB)

Observations Above graph shows simultaneous clients submitting the job. It is observed that on the x-axis the max number of jobs submitted is 6. It can be clearly observed that

Hadoop-CC shows better results as compared to default Hadoop configuration. It can also be observed that execution time of Hadoop-CC is consistently low.

Explanation The reason for only having 6 max jobs in parallel is due to infrastructure and memory limitations. The machines could not launch more than 6 JVMs limiting the experiment to run only 6 jobs at a time. The execution time is low due to the reason that data is available in cache causes more data-local tasks which causes overall decrease in the job execution time. Line graph and bar graph both taken together help us analyze the results better.

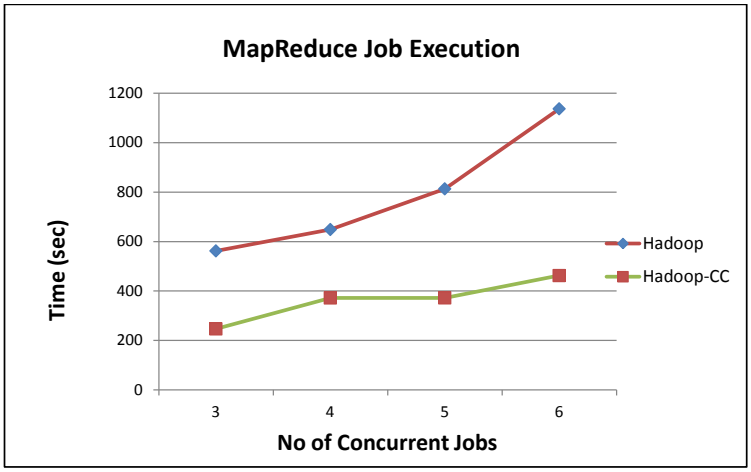

Figure 9.10: Multiple Job Execution Line Graph(File Size 500 MB)

Results It proves that collaborative caching helps in the overall decrease of the job execution time of the Hadoop system.

Chapter 10

Conclusions

The motivation to take this as the thesis work was due to the fact that Hadoop being widely used and to be able study in depth such widely used system and MapReduce framework which is used for huge data analysis and transformations.

In this thesis work, in depth study of Hadoop Architecture and its behavior was carried out. It's behavior was studied by performing initial experiments and analysis. Drawbacks were evaluated as well. After evaluating the architecture, a new architecture was proposed which is named as Hadoop-CC(Collaborative Caching). The architecture aimed at improving the overall MapReduce job execution time and efficiency of the system. This was done through the mechanism of collaborative caching where data is served from local caches as well as remote caching. This was combined with prefetching and caching reference. NameNode's response to client was modified to send cached locations as well to the non cached locations of DataNode. NameNode maintains the Global Cache Image which is image of location of all the cached blocks on the cluster. This is mapping of cached block to DataNodes indicating this block is cached on these DataNodes. On the DataNode side, each of them have their dedicated cache managers which have responsibilities of caching data, replacement policy which is Modified-ARC and prefetching of references. Caching of the references also improved the overall system execution. The cache was divided into four sections recent, recent history, frequent, frequent history instead of maintaining just a single cache. This mechanism helped in better caching replacement. Hence overall, the job execution time decreased by a considerable amount, efficiency of the system increased and

there were more data-local jobs scheduled.

10.1 Future Work

- A mechanism where we can plug in any type of caching mechanism in to the Hadoop system and parameter to define which files to cache.
- Responsibility of NameNode can be shared by multiple NameNodes such that each have their own namespace.
- Attempt can be made to focus more on data-local jobs execution.
- Advanced prefetching and collaborative caching techniques can be designed for Hadoop.

10.2 Limitations

- The thesis has been only tested for dataset ranging 10MB-1GB. It has not been tested for very large datasets.
- It has not been tested for all the real-world scenario.
- Moreover due to memory limitations, cache hit rates can only be tested for 350 MB - 650 MB.
- For testing multiple client job execution, due to memory limitation 6 parallel jobs.

Bibliography

[1] http://hadoop.apache.org/docs/r0.20.2/hdfs_design.html

[2] Dhruba Borthakur, et.al , Apache Hadoop Goes Realtime at Facebook, Proceedings of the 2011 International conference on Management of Data, SIGMOD 11. ACM, pp. 1071-1080.

[3] http://en.wikipedia.org/wiki/Apache_Hadoop

[4] http://www.slideshare.net/cloudera/mr-perf

[5] Christer A. Hansen. "Optimizing Hadoop for the cluster". University of Troms, Norway

[6] Dawei Jiang, Beng Chin Ooi, Lei Shi Sai Wu, "The Performance of MapReduce: An In-depth Study", Proceedings of the VLDB Endowment, Vol. 3, No. 1, 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.

[7] A. W. D. B. S. K. S. S. G. Ananthanarayanan, A. Ghodsi and I. Stoica. "Pacman: Coordinated memory caching for parallel jobs", In NSDI, 2012.

[8] David Leong, Collaborative Object Caching for Heterogenous OSD Clusters, Proceedings of the 2007 Computer Science and IT Education Conference, pp.425-436.

[9] Megiddo, N. et.al Outperforming LRU with an Adaptive Replacement Cache, Computer, IEEE, April 2004, pp.58-65

[10] Tom White, Hadoop: The Definitive Guide, OReilly Media, Inc., June 2009.

[11] Konstantin Shvachko, Hairong Kuan, Sanjay Radia and Robert Chansler, The Hadoop Distributed File System Mass Storage Systems and Technologies (MSST), IEEE 26th Symposium , May 2010 , pp. 1-10.

[12] http://kazman.shidler.hawaii.edu/ArchDoc.html

[13] http://developer.yahoo.com/hadoop/tutorial/module4.html

[14] http://www.gutenberg.org/

[15] ftp://ftp.uni-duisburg.de/linux/filesys/Lustre/whitepaper.pdf

[16] ”Gurmeet Singh”, ”Puneet Chandra”, ”Rashid Tahir”, ”A Dynamic Caching Mechanism for Hadoop using Memcached”, https://wiki.engr.illinois.edu/download/attachments/197297260/ClouData-1st.pdf?version=1&modificationDate=1330747624000

[17] http://hadoop.apache.org/releases.html

[18] Eric Anderson et.al. ”New Algorithms for File System Cooperative Caching“, Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium, pp.437-440.

[19] http://bradhedlund.com/2011/09/10/understanding-hadoop-clusters-and-the-network/

[20] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Google, 2004.

[21] http://hadoop.apache.org/docs/r0.20.2/quickstart.html#Local

[22] http://www.aosabook.org/en/hdfs.html

Chapter 11

Code

11.1 CachedBlock

As ths name suggests CachedBlock represents a cached block. CachedBlock holds the data and, when available, metadata associated with a block.

```
public class CachedBlock

private Block block;

private byte[] data;

private byte[] metadata;
```

11.2 CacheManager

As the name suggests CacheManager is the cache controller component of Datanode. It takes care caching blocks and streaming blocks from cache as they are being accessed/requested. It is responsible for maintaing the cache using particular caching and eviction policies.

It also prepares Cached Block report when requested by the DataNode and holds a copy of Global Cache Image as sent by the NameNode via DataNode.

```
public class CacheManager

private final Cache recent;
private final Cache frequent;

private final Cache recentHistory;
private final Cache frequentHistory;
```

11.3 DataTransferProtocol

In order to facilitate reading blocks from cache and allow various elements to provide caching related status messages/signals to each other, a couple of new status fields are added to the Data Transfer Protocol as follows

```
# Signal a request to read block from cache
public static final int OP_READ_BLOCK_CACHED = 101;

# Notify, block not found in local as well as current
# copy of Global Cache Image
public static final int OP_STATUS_BLOCK_NOT_CACHED = 111;

# Notify, block not found in local cache, but is cached on other datanode
# based on current copy of Global Cache Image
public static final int OP_STATUS_BLOCK_CACHED_ELSEWHERE = 112;
```

Chapter 12

Manual

Setting up Hadoop Cluster:

- Install java 1.6.
- Download Ubuntu 10.04.4 LTS (Lucid Lynx) installation image
- Follow the Hadoop set up guide to set up the cluster
- After setting up the cluster refer run MapReduce Job.
- Refer to ”Hadoop Most Commonly used commands” to copy files to HDFS and run MapReduce job.
- For setting up your machine in pseudo distributed mode, refer to section ”Set up Pseudo Distributed Mode” in Hadoop Installation Guide.